Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Suhas Thejaswi Muniyappa

Scalable Parameterised Algorithms for two Steiner Problems

Master's Thesis
Espoo, July 16, 2017

Supervisor: Professor Petteri Kaski

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Suhas Thejaswi Muniyappa		
Title:	Scalable Parameterised Algorithms for two Steiner Problems		
Date:	July 16, 2017	**Pages:**	viii + 104
Major:	Computer Science	**Code:**	T-110
Supervisor:	Professor Petteri Kaski		

In the Steiner Problem, we are given as input (i) a connected graph with nonnegative integer weights associated with the edges; and (ii) a subset of vertices called terminals. The task is to find a minimum-weight subgraph connecting all the terminals. In the Group Steiner Problem, we are given as input (i) a connected graph with nonnegative integer weights associated with the edges; and (ii) a collection of subsets of vertices called groups. The task is to find a minimum-weight subgraph that contains at least one vertex from each group. Even though the Steiner Problem and the Group Steiner Problem are $\mathcal{NP}$-complete, they are known to admit parameterised algorithms that run in linear time in the size of the input graph and the exponential part can be restricted to the number of terminals and the number of groups, respectively.

In this thesis, we discuss two parameterised algorithms for solving the Steiner Problem, and by reduction, the Group Steiner Problem: (a) a dynamic programming algorithm presented by Dreyfus and Wagner in 1971; and (b) an improvement of the Dreyfus–Wagner algorithm presented by Erickson, Monma and Veinott in 1987 that runs in linear time in the size of the input graph. We develop a parallel implementation of the Erickson–Monma–Veinott algorithm, and carry out extensive experiments to study the scalability of our implementation with respect to its runtime, memory bandwidth, and memory usage. Our experimental results demonstrate that the implementation can scale up to a billion edges on a single modern compute node provided that the number of terminals is small. For example, using our parallel implementation a Steiner tree for a graph with hundred million edges and ten terminals can be found in approximately twenty minutes. For an input graph with one hundred million edges and ten terminals, our parallel implementation is at least fifteen times faster than its serial counterpart on a Haswell compute node with two processors and twelve cores in each processor. Our implementation of the Erickson–Monma–Veinott algorithm is available as open source.

Keywords:	algorithm engineering, Dreyfus–Wagner algorithm, edge-linear time algorithms, group Steiner problem, Erickson–Monma–Veinott algorithm, parameterised algorithms, Steiner problem
Language:	English

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Petteri Kaski, for his guidance, careful review and helpful comments throughout the thesis work. Without his wisdom, help and encouragement this thesis would not be as it is today.

I would like to thank my friend Abdulmelik Mohammed for his useful insights throughout the process of writing this thesis and also for interesting discussions on complexity theory.

I would like to acknowledge the use of computational resources available via "Science-IT" project at Aalto University School of Science and via CSC– the Finnish IT center for Science, for the experiments carried out in this thesis.

Last but not least, many thanks are due to my most amazing parents who have been extremely supportive during the whole process. I am greatly indebted to my brother for his support and believing in what i do. And Shruthi, thank you for your encouragement and help at the times of failure; which made me recover fast.

Espoo, July 16, 2017

Suhas Thejaswi Muniyappa

Contents

Chapter 1

Introduction

The formal birth of graph theory is believed to be on August 26, 1735: the day Leonhard Euler (1707–1783) [39] presented his historic paper proving the infeasibility of traversing Königsberg's seven bridges crossing each bridge exactly once. However, references to informal usage of graph theory dates back to as early as twelfth century in Indian literature. Familysearch.org [40] and Shoumatoff [111] have reported that, the Pandits of Haridwar, Kurukshetra and other Hindu pilgrimages have been building family trees to trace ancestry and marriage records. In many places these records trace family history for over twenty prior generations stretching across nearly a millennium from 1194 to 2015. Since then graph theory has developed into an extensive and popular branch of mathematics which has been applied to many problems in computer science, bioinformatics, biology, chemistry, and other scientific and not-so-scientific areas.

Many real-world problems can be translated into graph-theoretic problems and thereby a non-trivial graph algorithm yields a better solution than an ad-hoc solution. There are well-known graph problems modelled from real-world problems such as vertex cover, shortest path, travelling salesman, facility location, matching, and many more. One such problem motivated from the real-world scenarios is the *Steiner problem*. An instance of an informal reference to the Steiner problem dates back to the period of the Indian emperor Chandragupta Maurya (321–297 BC). After succeeding in conquering and subjugating most of the provinces in Indian subcontinent, Chandragupta Maurya wanted to build a road network to connect the five provincial capitals of his empire with several parts of Western Asia to improve trade and further strengthen his empire. From an economic point of view these newly established roads should be as short as possible and should be built on the basis of existing roads. Deciding which roads should be selected was a prob-

lem for Chandragupta Maurya. We shall see that Chandragupta Maurya's problem can be modelled as a Steiner problem in graphs and the solution to the problem has a core that is a Steiner tree (see Chapter 3). However, the problem instance is merely an informal reference to the Steiner problem in history and it is not considered as the origin of the problem. The road network developed during Chandragupta Maurya's period laid the foundation for possibly the longest road in the Indian subcontinent known as the *Grand Trunk Road*. For the history of the grand trunk road we refer the reader to a book by Sarkar [108] and an article in Dhaka Tribune [29].

In the subsequent paragraphs we present a survey of the history of the Steiner problem and our survey is based on a report by Brazil *et al.* [13]. Most mathematicians believe that the first reference to the Steiner problem was made by Pierre Fermat (1607–1665) in 1643 in the context of finding a point whose sum of the distances from three other points in a plane is minimised and it is known as the *Fermat problem*. The Fermat problem is closely related to one of the forty known variants of the Steiner problem known as the *Euclidean Steiner problem*. The minimum distance point is called the *Toricelli point* or *Fermat-Torricelli point*, and it is named after Evangelista Torricelli (1608–1647) who proposed the first geometric solution for finding such a minimum-distance point in 1646 [118, 119]. A generalisation of the Fermat problem which seeks a point in a plane whose sum of the distances from n given points is minimum was proposed by Thomas Simpson in the book *Fluxions* in 1750.

The history of the Steiner problem is generally not well understood, especially prior to the 1930s. In many occasions the problem has been completely forgotten and then rediscovered many years later. In 1934, Jarnik and Kössler [71] presented the shortest-interconnection network problem to find the shortest network spanning n points in a plane. However, Jarnik and Kössler made no reference to the Fermat problem. Later in 1941, Courant and Robbins [25] established the connection between the Fermat problem and the shortest interconnection network problem, and coined the name the *Steiner problem* which is named after the famous German mathematician Jacob Steiner (1796–1863), although Steiner's contribution to the Steiner problem is unclear. In 1961, Melzak [94] showed that the shortest-interconnection network could be found in a finite number of steps and he also established many basic properties of the shortest-interconnection network. Later in 1968, Gilbert and Pollak [56] referred the shortest-interconnection network as the *Steiner minimal tree*, since the shortest connected network in a graph is a tree. They also extended the Steiner problem to d-dimensional spaces and studied a probabilistic version of the problem. The Steiner problem is one of

the twenty-one original $\mathcal{NP}$-complete problems discussed by Karp [77, 78]. Since the 1960s an increasingly sophisticated mathematical theory of minimal networks has been developed around the Steiner problem building on a combination of techniques from combinatorics, graph theory, geometry and algebra. The interest in the Steiner problem stems not only from the challenge it presents mathematically but also from its range of potential applications in the areas such as communication and infrastructure networks, physical chip design, social and data mining, and many more. Many influential mathematicians and computer scientists including E. J. Cockayne, M. R. Garey, D. S. Johnson, R. M. Karp, J. B. Kruskal, C. H. Papadimitriou and R. C. Prim have investigated questions that relate to the Steiner problem.

Due to its extensive application areas, as many as forty variants of the Steiner problem have been introduced to date and the *group Steiner problem* is one among these forty known variants (Hauptmann and Karpiński [59]). To the best of the authors knowledge, the first formal reference to the group Steiner problem was made by Cockayne and Melzak [22] in 1969. They introduced the Euclidean version of the group Steiner problem in two-dimensional planes. The problem was further studied by Hwang and Weng [67] in 1986. To the best of the authors knowledge, the group Steiner problem in graphs was first studied by Riech and Widmayer [104] in 1989, they modelled an application with regard to the layout of integrated circuits into the group Steiner problem and further formulated two heuristics. Ihler, Reich and Widmayer [68] in 1995 gave computational results and showed that the problem is $\mathcal{NP}$-hard by reducing the 3-SAT problem to the group Steiner problem.

The group Steiner problem arises in multiple applications. For example, in VLSI circuit design where the circuit modules on a VLSI chip have multiple connection points which can be rotated or flipped to reduce the connection distances (Reich and Widmayer [104]). The connection points correspond to a group of vertices in the group Steiner problem formulation, where there is a single connection point for each circuit module. The group Steiner problem also has a number of applications in the database and data-mining community. We can formulate the keyword-search problem in relational databases as the group Steiner problem. More precisely, a relational database can be modelled as a graph; where each vertex denotes a tuple, each edge represents a foreign-key reference between two tuples and the weight associated with an edge represents the strength of their relationship. The keyword-search problem aims to find a set of connected tuples that cover all the given keywords with minimum total weight on the induced edges. The optimal solution for the keyword-search problem is a minimum-weight connected tree covering all the keywords, and therefore the problem is an instance of the group Steiner prob-

lem. Many researchers in data-mining community have been using the Steiner tree and the group Steiner tree search techniques to solve numerous data-mining problems including community detection problem (Chiang *et al.* [20]; Sozio and Gionis [113]), keyword search problem (Coffman and Weaver [23]; Ding *et al.* [32]), team formation problem (Anagnostopoulos *et al.* [2]; Lappas, Liu and Terzi [88]), and many more. However, knowing the fact that the Steiner and group Steiner problems are $\mathcal{NP}$-complete, many solutions developed employ approximation algorithms (Bhalotia *et al.* [9]; Ding *et al.* [32]; He *et al.* [60]; Kacholia *et al.*[75]; Rozenshtein *et al.* [106]).

From the algorithm design perspective, the Steiner and group Steiner problems exhibit both formidable hardness and ease of scalability, meaning that even though the problems are $\mathcal{NP}$-complete they admit algorithms that run in polynomial time in the size of the host graph and the exponential complexity can be isolated to a parameter independent of the size of input graph (number of terminals and number of groups, respectively). Furthermore, It is possible to solve the Steiner and group Steiner problems in edge-linear time using parameterised algorithms (Erickson, Monma and Veinott [38]; Haugardy, Silvanus and Vygen [63]). The decision variant of the Steiner and group Steiner problems are $\mathcal{NP}$-complete (Karp [77]; Ihler, Reich and Widmayer [68]), and exact algorithms for $\mathcal{NP}$-complete problems have superpolynomial time complexity, unless $\mathcal{P} = \mathcal{NP}$. Yet, many distinct exact algorithms have been developed for solving the Steiner and group Steiner problems. We review the related work and the algorithms developed to date in Chapter 5. In this thesis, we discuss two parameterised algorithms for solving the Steiner problem, and by reduction, the group Steiner problem: (i) a dynamic-programming algorithm presented by Dreyfus and Wagner [34] in 1971; and (ii) an improvement of the Dreyfus–Wagner algorithm presented by Erickson, Monma and Veinott [38] in 1987 that runs in linear time in the size of the input graph. Our primary objective of this thesis is to present a parallel implementation of the Erickson–Monma–Veinott algorithm which can scale to large graphs. In addition to this, we perform experiments to verify the scalability of our implementation with respect to its runtime, memory bandwidth and memory usage.

The remaining chapters of this thesis are organised as follows: we will begin with a brief introduction to the graph-theoretic preliminaries, definitions and trivial proofs of graph theory in Chapter 2. In Chapter 3, we will introduce the Steiner problem and the group Steiner problem in graphs, and present their complexity results. For completeness and clarity, we provide a brief introduction to the complexity theory. In Chapter 4, we review the advances in priority queues and present pseudo-code for implementing the

priority queues using Fibonacci heap. We also discuss an edge-linear time algorithm based on the visit-and-label approach introduced by Dijkstra for solving the shortest path problem, which is trivial for designing an edge-linear time algorithm for the Steiner problem in graphs. In Chapter 5, we review the existing work and summarise the algorithms presented for the Steiner and group Steiner problems. We present two parameterised algorithms for the Steiner problem, one of which is the Dreyfus–Wagner algorithm [34], and the other is the Erickson–Monma–Veinott algorithm [38]. Furthermore, we discuss a linear-time reduction introduced by Voß [122] for solving the group Steiner problem as the Steiner problem. In Chapter 6, we develop a parallel implementation of the Erickson–Monma–Veinott algorithm to solve the Steiner problem, and by reduction, to solve the group Steiner problem. We present an implementation of Dijkstra's algorithm for finding a shortest path. Additionally, we present priority-queue implementations using binary and Fibonacci heap data-structures. In Chapter 7, we discuss the experimental results of our implementation of the Erickson–Monma–Veinott algorithm and Dijkstra's algorithm. Finally, we conclude this thesis in Chapter 8.

Chapter 2

Graph-theoretic preliminaries

Graph theory is a powerful abstraction tool and provides a framework for modelling a large set of computer-science problems including many real-world problems. The knowledge of graph-theoretic preliminaries is essential for understanding the advanced topics in combinatorial optimisation of graph-theoretic problems. In this chapter, we give a brief introduction to basic graph-theoretical terminologies, basic definitions in graph theory and familiarise with the notations used in this thesis. In addition, all the definitions will be brought to life by a number of simple yet fundamental propositions. In this thesis, we follow the notation and terminology used in Diestel [30]. For the undefined terms, we refer the reader to introductory books in graph theory by Diestel [30] and West [128].

2.1 Graphs

A graph G is an ordered pair (V, E) where V is a finite set of *vertices* (or *nodes*) and $E \subseteq \{\{u, v\} : u, v \in V\}$ is a set of unordered pair of distinct vertices called *edges*, and the graph is denoted as $G = (V, E)$. For an edge $e = \{u, v\}$ we say u and v are *end-points* of e.

The vertex set of a graph $G = (V, E)$ is referred as $V(G)$ and its edge set as $E(G)$. The *order* of G is the number of vertices in G and it is denoted as $n = |V|$; the *size* of G is the number of edges in G and it is denoted as $m = |E|$. An example of a graph is shown in Figure 2.1.

An edge e is *incident* to a vertex v if v is one of the end points of e. A vertex u is said to be *adjacent* to vertex v if u and v are end points of an edge

$e \in E$. The set of vertices adjacent to v is called the set of *neighbours* of v and it is denoted as $\mathsf{adj}(v)$. The *degree* of a vertex v is the number of vertices adjacent to v and it is denoted as $\mathsf{deg}(v) = |\mathsf{adj}(v)|$. The graph is *d-regular* if all the vertices in graph have degree d. A *subgraph* $G' = (V', E')$ of a graph $G = (V, E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq E$. If G' contains all the edges $\{u, v\} \in E$ with $u, v \in V'$ then G' is an *vertex induced subgraph* of G and it is denoted as $G' = G[V']$.

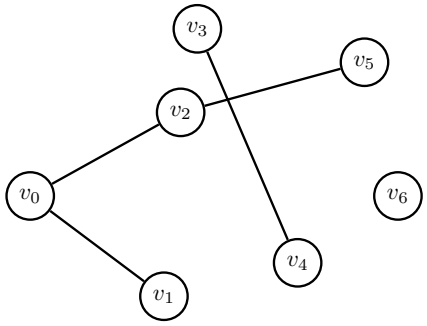


Figure 2.1: A graph on vertex set $V = \{v_0, v_1, \ldots, v_6\}$ with edge set $E = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_2, v_5\}, \{v_3, v_4\}\}$; the order of graph is $|V| = 7$ and its size is $|E| = 4$.

2.2 Weighted graphs

A *weighted graph* (or *network*) N is a triple (V, E, w) with an underlying graph (V, E) and a cost function $w : E \to \mathbb{Z}_{\geq 0}$ that maps each edges $e \in E$ to a *value* (or *weight*). An example of a weighted graph is shown in Figure 2.2.

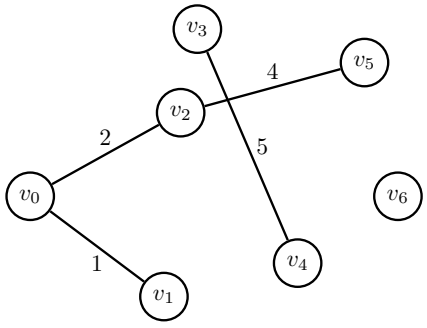


Figure 2.2: A weighted graph on vertex set $V = \{v_0, v_1, \ldots, v_6\}$ with edge set $E = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_5, v_2\}, \{v_4, v_3\}\}$ and a cost function mapping $w(\{v_0, v_1\}) \to 1, w(\{v_0, v_2\}) \to 2, w(\{v_2, v_5\}) \to 4, w(\{v_3, v_4\}) \to 5$.

2.3 Walk, trail, path, connectivity

A *walk* in a graph $G = (V, E)$ is a sequence $\mathcal{W} = (v_0, e_0, v_1, \ldots, v_k)$ of alternating vertices and edges for some $k \geq 0$ where vertices $v_0, \ldots, v_k \in V$ and edges $e_0, \ldots, e_{k-1} \in E$ such that for each $i = 0, 1 \ldots, k-1$ it holds that $e_i = \{v_i, v_{i+1}\} \in E$. The vertex set of $\mathcal{W}$ is denoted by $V(\mathcal{W})$ and the edge set is denoted by $E(\mathcal{W})$. The vertices v_0 and v_k are called the *initial* and *terminal* vertices of $\mathcal{W}$. If $v_0 = v_k$ then $\mathcal{W}$ is *closed*; otherwise it is *open*. The *length* of $\mathcal{W}$ in a graph is the number of edges in $E(\mathcal{W})$ and it is denoted as $W(\mathcal{W}) = k$. The *length* of $\mathcal{W}$ in a network is the sum of weights of edges in $E(\mathcal{W})$ that is $W(\mathcal{W}) = \sum_{i=0}^{k-1} w(e_i)$.

A walk with distinct edges is called a *trail*. A *cycle* is a trail with at least two vertices with no repeated vertices, except the initial and terminal vertex. A graph with no cycles is *acyclic*; otherwise it is *cyclic*.

A *path* is an open walk with distinct vertices. Two paths P and P' between vertices u, v are *distinct* if there exists at least one vertex x such that $x \in V(P)$ and $x \notin V(P')$ or vice versa. A *subpath* S of a path $P = \{v_0, e_0, v_1, \ldots, v_k\}$ is a path between any two vertices $v_i, v_j \in V(P)$ such that $V(S) \subset V(P)$ and $E(S) \subset E(P)$.

A graph is *connected* if there exists a path between every pair of vertices; otherwise it is *disconnected*. A vertex v *is reachable from* vertex u if there exists a path from u to v. The relation *is reachable from* is reflexive, transitive and symmetric. The *graph distance* between two vertices $u, v \in V$ in a graph G is the length of a shortest path connecting them and it is denoted as $d_G(u, v)$.

Proposition 2.1. In a graph the union of two distinct paths between any two distinct vertices contains a cycle.

Proof. Let $P = (v_0, e_0, v_1, \ldots, v_m)$ and $P' = (v'_0, e'_0, v'_1, \ldots, v'_n)$ be two distinct paths such that their end vertices satisfy $v_0 = v'_0$ and $v_m = v'_n$. Let us choose an intermediate vertex v_i such that $v_j = v'_j$ for all $0 \leq j \leq i$ and $v_{i+1} \neq v'_{i+1}$, such an intermediate vertex always exists otherwise $P = P'$. Let k be the minimum index of a vertex for some $i < k \leq m$ such that vertex v_k is identical to vertex v'_p for some index $i < p \leq n$; such an index must exist otherwise we contradict $v_m = v'_n$. Let S and S' be the subpaths between v_k and v'_p along the paths P and P', respectively, then S and S' are disjoint. The union of S and S' clearly forms a cycle. If there exists an index x such that vertex v_x is identical to vertex v'_q for some index $x < k$, it contradicts the minimality of k. If $x = k$ then $v_x = v'_p$. □

Proposition 2.2. In a connected network $N = (V, E, w)$ there exists a path between any two vertices with length at most $W(N) = \sum_{e \in E} w(e)$.

Proof. Let N be a connected network. Since N is connected there exists a path between any two distinct vertices in N. For the sake of contradiction, let us assume that P be a path between any two vertices in N such that $W(P) > W(N)$. Clearly, this is possible if at least one edge is repeated in P. On the other hand, it contradicts our initial assumption that P is a path and no edges are repeated. Hence in a connected network there exists a path between any two vertices with length at most $W(N)$. □

Corollary 2.3. In a connected network $N = (V, E, w)$ there exists no path with length greater than $W(N) = \sum_{e \in E} w(e)$.

2.4 Trees

An acyclic graph is a *forest*. A connected forest is a tree. A vertex with degree one in a tree is called a *leaf vertex*. A vertex with degree greater than one is called an *internal vertex*. A tree is *trivial* if it consists of a single vertex. An example of a forest and a tree is shown in Figure 2.3.

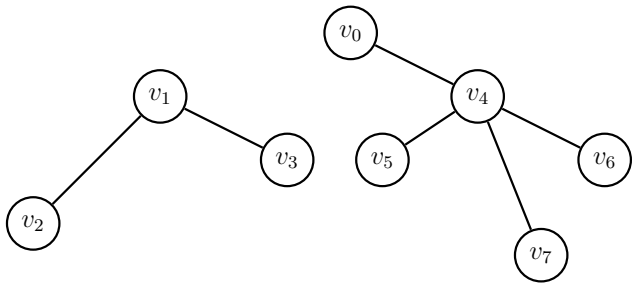
A tree is *rooted* if a special vertex is singled out and the special vertex is called the *root vertex*. In a rooted tree, the *parent* of a vertex is the vertex connected to it on the path to the root; every vertex except the root vertex has an unique parent. A *child* of a vertex v is a vertex for which v is the parent. The set of vertices having the same parent in a rooted tree is called a set of *siblings*.

The length of a longest path in a tree is its *diameter*. The *height* of a rooted tree is the length of longest path from root vertex to any leaf vertex. The *depth* of a vertex in a rooted tree is the length of the path from the vertex to the root vertex of the tree; the *level* of a vertex in a rooted tree is one more than the depth of the vertex, more precisely level of v is 1+ depth of v. The *rank* of a vertex v in a tree is the number of children of v and it is denoted as $\mathsf{rank}(v)$.

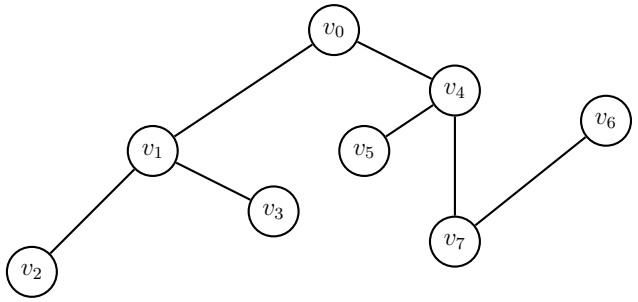
Proposition 2.4. Every tree with at least two vertices has at least one leaf vertex.

Proof. Let us consider a tree with at least two vertices. Since a tree is a connected acyclic graph, every vertex has at least degree one. For the sake

of contradiction suppose a tree has no leaf vertices. Start a walk from some arbitrary vertex, since the graph has no leaf vertices each vertex has degree two or more. So we can always leave a vertex by an edge different from the one by which we reached and there are no dead ends. Nevertheless, there are only finitely many vertices in graph, so we will end up visiting a vertex which is previously visited and it forms a cycle. However, this contradicts the definition of a tree. Hence, there exists at least one leaf vertex in a tree. □



(a)



(b)

Figure 2.3: An example of (a) a forest; and (b) a tree.

Proposition 2.5. A connected graph is a tree if and only if there exists an unique path between any two vertices in the graph.

Proof. Let $G = (V, E)$ be a graph with exactly one path between every pair of vertices. So that makes G connected. Let us assume that G has a cycle on vertex $v \in V$ then there exists an intermediate vertex u in the cycle such that there are at least two distinct paths from u to v. However this contradicts our initial assumption that every pair of vertices in G has exactly one path. Hence G is a connected graph without cycles which makes G a tree.

Conversely, let $G = (V, E)$ be a tree. By the definition of a tree G is a connected acyclic graph. So there is at least one path between every pair

of vertices in G. For the sake of contradiction let us assume that there are more than one distinct paths between vertices $u, v \in V$. However, the union of two distinct paths between u and v contains a cycle, this contradicts our assumption that G is a tree (see Proposition 2.1). Hence, there exists an unique path between any two vertices in a tree. □

Proposition 2.6. A tree with $n \geq 1$ vertices has exactly $n - 1$ edges.

Proof. We prove the result using induction. The result is true for a tree with $n = 1$ and $n = 2$ vertices since the number of edges in a tree with one vertex is zero and the number of edges in a tree with two vertices is one. Let us assume that the result is true for all trees less than or equal to $n - 1$ vertices. Consider a tree $T = (V, E)$ with $n > 2$ vertices and an arbitrary leaf vertex $v \in V$ (from Proposition 2.4, such a leaf vertex always exists in a tree). By deleting v from T, we obtain tree T' with $n - 1$ vertices; however from induction hypothesis, all trees with $n - 1$ vertices have $n - 2$ edges. Since v is a leaf vertex there is exactly one edge incident to v in T thus T has one more edge than T'. Hence the number of edges in a tree with n vertices is $n - 1$. □

Corollary 2.7. A tree with $n - 1 \geq 0$ edges has exactly n vertices.

Proposition 2.8. A connected graph with $n \geq 1$ vertices and $n - 1$ edges is a tree.

Proof. Let G be a connected graph with $n \geq 1$ vertices and $n - 1$ edges, then we show that G is acyclic. For the sake of contradiction, let us assume that G is a cyclic graph and has at least one cycle. So remove an edge from a cycle at a time and continue the process until the resultant graph is a tree (connected acyclic graph). Let T be the tree formed after removing edges from G then clearly T has less number of edges than G, which contradicts the proof of Proposition 2.6. Hence G is acyclic and therefore it is a tree. □

2.5 Binary trees

A *binary tree* is a rooted tree in which each vertex has at most two children. This means that every vertex in a binary tree has degree at most three and it is often convenient to designate each child of a vertex as its *left* or *right* child. A binary tree is *proper* if every internal vertex has two children. More precisely, in a proper binary tree every vertex has zero or two children. An example of a binary tree is shown in Figure 2.4.

A binary tree is *perfect* if all the levels are completely filled or equivalently, all leaf vertices are at the same depth. A binary tree is *complete* if every level except possibly the last level is completely filled and all leaf vertices are as far left as possible. An example of a complete-binary tree is shown in Figure 2.5.

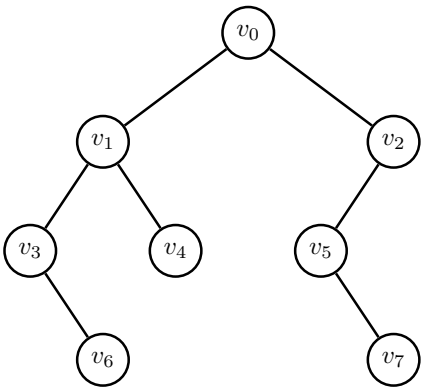


Figure 2.4: A binary tree on vertex set $\{v_0, v_1, \ldots, v_7\}$ with root vertex v_0 and leaf vertices v_4, v_6, v_7. Each vertex in a binary tree can be designated a left and a right child; for example, the left child of vertex v_0 is v_1 and the right child is v_2. The height of the tree is three.

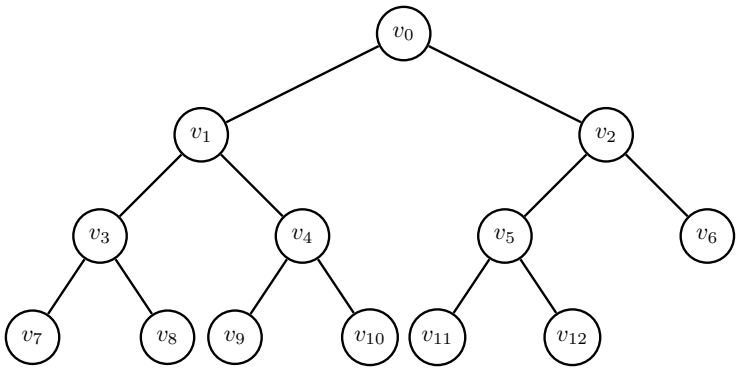


Figure 2.5: An example of a complete-binary tree with vertex set $\{v_0, v_1, \ldots, v_{12}\}$, root vertex v_0 and leaf vertices $v_6, v_7, \ldots, v_{12}$. Every level except the last level is completely filled and all leaf vertices are as far left as possible.

2.6 Binomial trees

A *binomial tree* of rank $k \geq 0$ is defined recursively as follows: a binomial tree of rank $k = 0$ consists of a single vertex; and a binomial tree of rank $k > 0$

consists of a root vertex with k binomial sub-trees of ranks $0, 1, \ldots, k-1$. A binomial tree of order k has exactly 2^k vertices and height k. A *binomial forest* is a collection of binomial trees. An example of a binomial tree is shown in Figure 2.6.

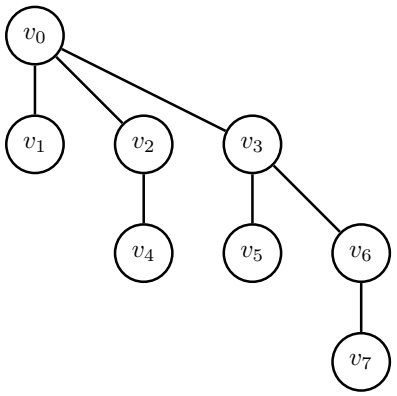


Figure 2.6: An example of a binomial tree on vertex set $\{v_0, v_1, \ldots, v_7\}$ with root vertex v_0 and leaf vertices v_1, v_4, v_5, v_7. In addition, the order of root vertex v_0 is 3, and it has children v_1, v_2 and v_3 with orders $0, 1$ and 2, respectively.

In this thesis, we only consider loopless, connected and undirected graphs unless explicitly stated otherwise.

Chapter 3

The Steiner problem in graphs

The (*minimum*) *Steiner problem* is a classical combinatorial optimisation problem and the historical background of the problem traces back to Pierre Fermat who proposed the Euclidean Steiner problem: find a point p in a triangle such that the sum of distances from p to three vertices of the triangle is minimum, this problem is also known as the *Fermat problem* and the point p is called the *Torricelli point* or the *Fermat-Torricelli point*. The Steiner problem is a combinatorial variant of the Fermat problem. The Steiner problem in graphs is also referred as the *Steiner tree problem* or the *minimum Steiner tree problem*.

In this chapter, we will introduce the Steiner problem and the group Steiner problem in graphs, and further discuss their computational complexity. Let us begin our discussion with an introduction to computational complexity theory and the concept of reducibility among computational problems.

3.1 Computational complexity

Computational complexity theory is a branch of theoretical computer science which deals with classifying and relating the computational problems according to their inherent difficulty. In this section, we recall the basic definitions in complexity theory which are relevant for understanding the $\mathcal{NP}$-completeness proof presented in the later sections of this chapter. The study of complexity theory deal with a wide range of aspects related to the computational problems and our interest only lies in proving the $\mathcal{NP}$-completeness result of the Steiner and group Steiner problems. To keep things simple this section provides only an overview of the complexity theory and does not cover all

the concepts in a great detail. Moreover, we only define terms and notations which are relevant for the purpose of this thesis and it closely resembles the terminology used by Garey and Johnson [53]. For more-advanced topics in complexity theory we refer the reader to books by Arora and Barack [4], and Papadimitriou [101].

Let $\Sigma = \{0, 1\}$ and let Σ^* denote the set of all finite strings of symbols from Σ. An *instance* $I \in \Sigma^*$ of a problem is a string. A set $\mathcal{I} \subseteq \Sigma^*$ of strings is *recognisable in polynomial time* if there exists a constant $c \in \mathbb{N}$ and an algorithm $\mathcal{A}$ that stops for every string $I \in \Sigma^*$ after at most $O(|I|^c)$ steps and returns ACCEPT if $I \in \mathcal{I}$ and REJECT if $I \in \Sigma^* \setminus \mathcal{I}$. For the purpose of this thesis, we only consider instances recognisable in polynomial time.

A decision problem Π is described as a tuple $(\mathcal{I}, Sol)$ where $\mathcal{I} \subseteq \Sigma^*$ is a set of instances that is recognisable in polynomial time and Sol is a mapping that associates with each instance $I \in \mathcal{I}$ to a set $Sol(I) \subseteq \Sigma^*$ of solutions of I. An algorithm $\mathcal{A}$ is said to solve a decision problem $\Pi = (\mathcal{I}, Sol)$ if $\mathcal{A}$ stops for all instances of $I \in \mathcal{I}$ and returns ACCEPT if $Sol(I) \neq \emptyset$ and REJECT otherwise.

A decision problem Π belongs to class $\mathcal{P}$ if there exists an algorithm which solves Π in polynomial time. This means that the class $\mathcal{P}$ consists of all problems which can be solved in polynomial time. However, for many decision problems no polynomial-time algorithm is known. Nevertheless, some of these problems have a property which is not inherent to every decision problem: if presented with an instance $I \in \mathcal{I}$ of Π and a potential solution $s \in \Sigma^*$ such that $|s| \leq |I|^c$, $c \in \mathbb{N}$ then there exists a polynomial-time algorithm which verifies whether $s \in Sol(I)$. The decision problems with this property form the class *Non-deterministic polynomial time* or $\mathcal{NP}$. Put differently, non-deterministic polynomial time essentially means it is easy to verify a given solution but it need not be easy to find such a solution.

A decision problem $\Pi = (\mathcal{I}, Sol)$ is said to be *reducible* to another decision problem $\Pi^* = (\mathcal{I}^*, Sol^*)$ if there exists a function $f : \mathcal{I} \to \mathcal{I}^*$ such that for each $I \in \mathcal{I}$, $Sol(I) \neq \emptyset \iff Sol^*(f(I)) \neq \emptyset$. If f is computable in polynomial time then the reduction is polynomial. In this thesis, a reduction implicitly means a polynomial-time reduction unless explicitly stated otherwise.

To prove that a given problem Π is $\mathcal{NP}$-complete we follow the steps advised by Garey and Johnson [53] and the steps are listed as follows:

1. prove that Π is in $\mathcal{NP}$,
2. select a known $\mathcal{NP}$-complete problem Π' and reduce Π' to Π, and
3. prove that the reduction is a polynomial time transformation.

To prove that the Steiner problem in graphs is $\mathcal{NP}$-complete: First, we prove that the problem is in $\mathcal{NP}$. Second, we reduce the exact 3-cover problem to the Steiner problem. Finally, we argue that such a reduction is possible in polynomial time. Before proceeding to the $\mathcal{NP}$-completeness proof of the Steiner problem let us define the exact 3-cover problem.

3.2 The exact 3-cover problem

The *Exact 3-Cover Problem* is a well known $\mathcal{NP}$-complete problem and it is mentioned among the basic $\mathcal{NP}$-complete problems in Garey and Johnson [53]. The problem is also referred as the exact cover by 3-sets problem.

In the Exact 3-Cover Problem, we are given as input: (i) a set $X = \{x_1, x_2, \ldots, x_q\}$; and (ii) a collection $\mathcal{C} = \{C_1, C_2, \ldots, C_p\}$ of 3-element subsets of X. The task is to find a set $\mathcal{C}' \subseteq \mathcal{C}$ such that each element of X appears in exactly one member of $\mathcal{C}'$. The decision version of the exact 3-cover problem is stated as follows:

The Exact 3-Cover Problem (X3C)

Instance: A finite set $X = \{x_1, x_2, \ldots, x_q\}$ and a collection $\mathcal{C} = \{C_1, C_2, \ldots, C_p\}$ of subsets of X such that $C_i \subseteq X$ and $|C_i| = 3$.

Question: Does $\mathcal{C}$ contain an exact cover for X, that is a subset $\mathcal{C}' \subseteq \mathcal{C}$ such that every element of X occurs in exactly one member of $\mathcal{C}'$?

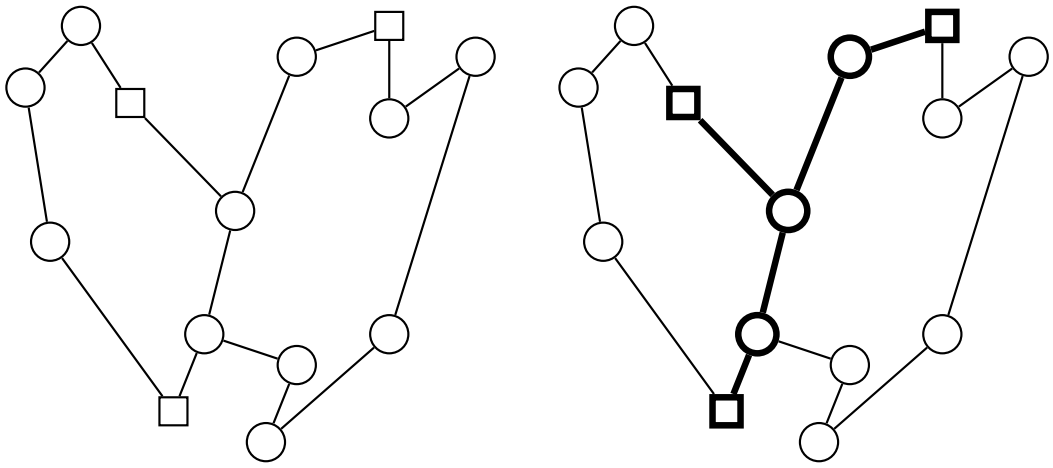
3.3 The Steiner problem in graphs

In the Steiner Problem in graphs, we are given as input: (i) a graph $G = (V, E)$; and (ii) a set $K \subseteq V$ of required vertices called *terminals*. The task is to find a tree T in G connecting all the terminals in K with minimum number of edges. The tree must contain all terminals; however it might also contain non-terminal vertices called the *Steiner vertices*. An example of the Steiner problem in graphs is shown in Figure 3.1. The decision version of the Steiner problem in graphs is stated as follows:

The Steiner Problem in graphs (SP)

Instance: A graph $G = (V, E)$ with a set of terminals $K \subseteq V$ and a bound $B \in \mathbb{Z}_{\geq 0}$.

Question: Does there exists a subgraph T connecting all vertices of K in G such that $W(T) \leq B$?



(a) (b)

Figure 3.1: An example of the Steiner problem in graphs: (a) A graph with four terminal vertices; the terminal vertices are denoted by rectangles and the non-terminal vertices are denoted by circles. (b) A Steiner tree for the given Steiner problem. The highlighted vertices and edges represent a Steiner tree for the given problem instance. The tree covers all the terminals and it also contain three non-terminal vertices called the Steiner vertices. The Steiner tree shown is the only possible solution for the given problem instance, however, we should keep in mind that it is possible to have more than one Steiner tree for a problem instance.

The decision variant of the Steiner problem in graphs is one of the twenty-one original $\mathcal{NP}$-complete problems stated by Karp [77, 78]. In 1977, Garey, Graham and Johnson [52] gave the complexity results of the Steiner problem for general planar point sets and showed that the problem is inherently at least as difficult as any of the $\mathcal{NP}$-complete problems. The $\mathcal{NP}$-completeness result we prove is an additional result for this thesis and it effectively serves as an excuse for not giving a polynomial-time algorithm for the problem. The proof presented is based on the work of Garey, Graham and Johnson [52] and it answers the question whether there exists an algorithm which solves the Steiner problem for every instance of the terminal set K in polynomial time and the answer is NO.

Lemma 3.1. The Steiner problem in graphs is $\mathcal{NP}$-complete.

Proof. Let us recall from Section 3.1 that to prove that a decision problem Π is $\mathcal{NP}$-complete, we should prove that: Π is in $\mathcal{NP}$ and there exists a polynomial-time reduction from a known $\mathcal{NP}$-complete problem Π' to Π.

Firstly, we prove that Steiner problem in graphs is in $\mathcal{NP}$. Let $\Pi = (G, K, B)$ be a Steiner problem instance which reserves a *yes* answer. Given a hypothetical solution $T \subseteq G$ we can verify in linear time that: T is a tree (connected graph with no cycles), T contain all terminals in K and the cost of tree $W(T) \leq B$. Hence, the Steiner problem in graphs is in $\mathcal{NP}$.

To prove that the Steiner problem in graphs is $\mathcal{NP}$-hard. We reduce the exact cover by 3-sets (X3C) problem to the Steiner problem (SP). Furthermore, we prove that such a reduction is possible in polynomial time (in fact the transformation is a linear-time reduction). Given an instance of X3C on a variable set $X = \{x_1, x_2, \ldots, x_{3q}\}$ and a collection of 3-element subsets $\mathcal{C} = \{C_1, C_2, \ldots, C_p\}$ of X. Our aim is to construct a graph $G = (V, E)$ with a terminal set K and bound B such that G contains a Steiner tree T if and only if the given X3C instance is satisfiable.

We construct a graph $G = (V, E)$, terminal set K and bound B such that,

$$
\begin{aligned}
V &= \{v\} \cup \mathcal{C} \cup X, \\
E &= \{\{v, C_i\} | C_i \in \mathcal{C}, i = 1, 2, \ldots, p\} \cup \\
&\quad \{\{C_i, x_j\} | x_j \in C_i, \text{ for each } C_i \in \mathcal{C}, i = 1, 2, \ldots, p,\ j = 1, 2, \ldots, 3q\}, \\
K &= \{v\} \cup X, \\
|V| &= p + 3q + 1, \\
|E| &= 4p, \\
B &= 4q.
\end{aligned}
$$

The graph G can be constructed in linear time, thus the reduction from X3C to SP is a linear-time transformation. The graph construction is illustrated in Figure 3.2.

To prove that their exists a Steiner tree with no more than $B = 4q$ edges if and only if there exists an exact cover for X3C instance with q elements. Let us assume that there exists an exact 3-cover $\mathcal{C}'$ such that $|\mathcal{C}'| = q$. As $\mathcal{C}'$ is an exact cover, it covers each variable $x_j \in X$ exactly once. By selecting a clause C_i we implicitly choose the edge $\{v, C_i\}$ and edges $\{C_i, x_j\}$ for all $x_j \in C_i$. Let T be the graph induced by selecting clauses in $\mathcal{C}'$. From the construction it is implicit that T is connected. There are $3q$ edges between $\mathcal{C}'$ and X, and q edges between v and $\mathcal{C}'$. Hence, the number of edges in T is $4q$. Since $\mathcal{C}'$ is an exact cover, $\mathcal{C}'$ has exactly q clauses and the terminal set has $3q + 1$ vertices. Hence the total number of vertices in T is $4q + 1$. Recall from Proposition 2.6, a connected graph with n vertices and $n - 1$ edges is a

tree and T spans over all the terminals in K. Hence, T is a Steiner tree with cost $W(T) = 4q$.

Conversely, let us assume that there exists a Steiner tree T with at most $B = 4q$ edges in G covering all the terminals K. Since T is a tree with at most $4q$ edges, from Proposition 2.5 T has at most $4q + 1$ vertices. From the definition of the Steiner tree T must cover all the terminals in K and $|K| = 3q + 1$. Consequently, there are at most q Steiner vertices in T covering all the terminals in K. Hence, the number of clauses selected in $\mathcal{C}'$ is at most q.

Let us assume that there exists $\mathcal{C}'' \subseteq \mathcal{C}$ such that $\mathcal{C}'' \neq \mathcal{C}'$ where $|\mathcal{C}''| = z$ and $z < q$, then clearly the number of terminals covered by the clauses C'' is at most $3z + 1 < 3q + 1$. As a result, there exists at least one vertex $x \in X$ which is not covered by T. However, this contradicts our assumption that T is a Steiner tree and covers all the vertices in $X \subset K$. Hence, there are exactly q clauses in $\mathcal{C}'$.

Let us assume that there exists at least one vertex $x_l \in X$ for some $l = 1, \ldots, 3q$ and covered by two clauses $C_i, C_j \in \mathcal{C}'$ such that $i \neq j$ then there exists at least one vertex in X which is not covered by $\mathcal{C}'$. However, this contradicts our assumption that T is a Steiner tree and covers all the vertices in $X \subset K$. Hence $\mathcal{C}'$ is an exact 3-cover with q clauses. □

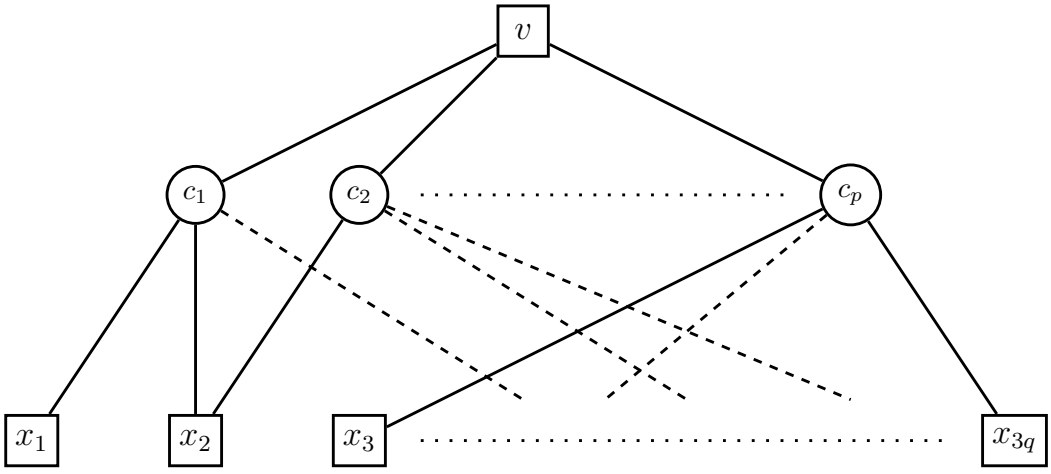


Figure 3.2: An illustration of the graph construction for the $\mathcal{NP}$-completeness proof of the Steiner problem in graphs. The terminal vertices are denoted by rectangles and non-terminal vertices are denoted by circles.

3.4 The group Steiner problem in graphs

In the Group Steiner Problem, we are given as input: (i) a graph $G = (V, E)$; and (ii) a collection $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ of subsets of V called *groups*. The task is to find a tree T connecting at least one vertex from each group $Q_i \in \mathcal{Q}$ with minimum number of edges. The groups need not be mutually-disjoint vertex sets. An example of the group Steiner problem in graphs is shown in Figure 3.3.

The decision version of the group Steiner problem in graphs is stated as follows:

The Group Steiner Problem in graphs (GSP)

Instance: A graph $G = (V, E)$, a collection $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ of groups and bound $B \in \mathbb{Z}_{\geq 0}$.

Question: Does there exists a tree T in G that connects at least one vertex from each group $Q_i \in \mathcal{Q}$ such that $W(T) \leq B$?

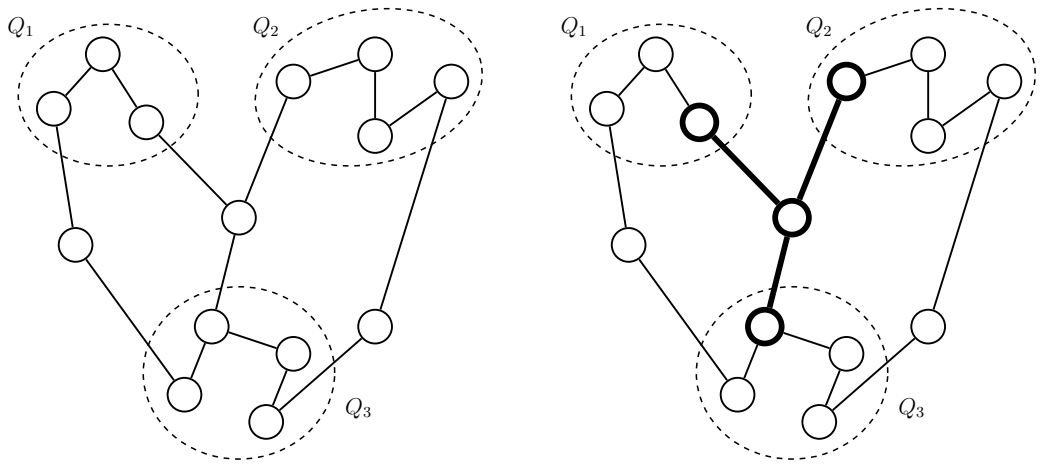


Figure 3.3: An example of the group Steiner problem in graphs: (a) A graph with three groups $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$; the vertices within each dotted region form a group. (b) The highlighted tree represent a group Steiner tree connecting at least one vertex from each group Q_1, Q_2 and Q_3. The group Steiner tree shown is the only possible solution for this problem instance, however we should keep in mind that it is possible to have more than one group Steiner tree for a given problem instance.

To the best of the authors knowledge, the group Steiner problem in graphs was introduced by Reich and Widmayer [104] in the context of wire-routing phase in physical VLSI designs. It is one of the forty known variants of the Steiner problem (Hauptmann and Karpiński [59]). Ihler, Reich and Widmayer [68] in 1995 showed that the problem is $\mathcal{NP}$-complete by reducing the 3-SAT problem to the group Steiner problem.

With Lemma 3.1 at hand, we can prove that the group Steiner problem in graphs is $\mathcal{NP}$-complete by considering the Steiner problem as a restricted case of the group Steiner problem with each group as a single vertex.

Theorem 3.2. *The group Steiner problem in graphs is $\mathcal{NP}$-complete.*

Proof. Every instance of the Steiner problem in graphs can be viewed as an instance of the group Steiner problem in graphs by simply regarding it as, each group has a single vertex (each group is a singleton). Consequently, the Steiner problem is a restricted version of the group Steiner problem and the $\mathcal{NP}$-completeness of the group Steiner problem follows by a trivial transformation from the Steiner Problem. From Lemma 3.1, the Steiner problem in graphs is $\mathcal{NP}$-complete. Thus, the $\mathcal{NP}$-completeness of the group Steiner problem follows directly from the $\mathcal{NP}$-completeness of the Steiner problem. □

Now that we have introduced enough terminology to define a Steiner problem, let us recall Chandragupta Maurya's problem to build a road transport network that connects the five provincial capitals of his empire with several parts of Western Asia and these newly established roads should be as short as possible and should be built on the basis of the existing roads (see Chapter 1). Actually, this problem can be transformed to the problem of finding a Steiner tree in network $N = (V, E, w)$. In this network, each city is represented by a vertex $v \in V$, each road link between any two cities form an edge $e \in E$ and the length of each road corresponds to the edge-weight $w(e)$. In addition, the provincial capitals and the major cities in Western Asia that need to be connected form the terminal set K.

Chapter 4

Data-structures and basic algorithms

The efficiency of an algorithm processing large-amount of data mainly depends on the use of appropriate and intelligent data-structures. Some instances of data manipulation involve drastic change in the size of the data-structure during the execution of algorithms. However, the time and space complexity of such large-amounts of data manipulation can be reduced by employing efficient data-structures. A priority queue is an example of one such data-structure which is extensively used in the field of computer science.

Priority queues are used in a wide range of applications including: job scheduling algorithms, discrete simulation languages, numerical analysis algorithms, sorting algorithms, graph algorithms, and many more (Brown [16]; Charters [19]; Floyd [41]; Gentleman [55]; Gonnet [57]; Jonassen and Dahl [74]; Williams [129]). However, the search for a better priority queue is motivated by its use in two fundamental computer-science problems: (i) the minimum spanning tree problem; and (ii) the shortest path problem. In our case it is the latter.

The Steiner problem in graphs has a sub-problem of finding a shortest path in graphs. More precisely, this is our motivation to study the shortest path problem and the data-structures to improve the time complexity of algorithms used for computing a shortest path. In particular, we focus on priority-queue implementation using Fibonacci heap, which is essential for designing an edge-linear time algorithm for the shortest path problem, furthermore, an edge-linear time algorithm for the Steiner problem in graphs.

In this chapter, we will review the advances in priority-queue implementa-

tion using different data-structures and discuss its use in Dijkstra's algorithm for finding a shortest path in graphs. In Section 4.1, we introduce priority queues and Sections 4.2 to 4.6 is a survey of priority-queue implementations using binary, binomial, Fibonacci and strict Fibonacci heaps. In addition, Section 4.4 provides pseudo-code for implementing priority queue operations using Fibonacci heap. In Section 4.5, we give a brief introduction to amortised complexity analysis. In particular, physicist's view of amortised analysis introduced by Sleator and Tarjan [114]. In Section 4.7, we introduce the shortest path problem in graphs. Finally, in Section 4.8, we discuss an algorithm based on visit-and-label approach introduced by Dijkstra for finding a shortest path in graphs.

4.1 Priority queues

A *priority queue* (or *heap*) is a data-structure for maintaining a set S of elements such that each element in S is associated a value called a *key* (or *priority*) (Cormen *et al.* [24]). The priority of an element e is denoted as $\mathsf{key}(e)$. In practice, two types of priority queues are used: a *minimum priority queue* (or *min-heap*) and a *maximum priority queue* (or *max-heap*). In a minimum (maximum) priority queue, always the element with low (high) priority is served before the element with high (low) priority. If a queue has more than one element with same priority then elements are served according to their order of occurrence in the queue. For the simplicity of our discussion, we associate the value of an element to its priority, whereas in practice, the priority and value of an element can be different. In what follows, the use of term priority queue or heap generally mean a minimum-priority queue unless explicitly stated otherwise.

A priority queue supports at least Insert, Delete, Extract-min and Decrease-key operations. These operations are briefly described as follows:

Insert:	add a new element.
Delete:	remove the element.
Extract-min:	remove and return the element with minimum priority.
Decrease-key:	change the priority of the element to a new smaller value.

A priority queue can be implemented using different data-structures ranging from a one-dimensional array to more complicated data-structures such as binomial and Fibonacci heaps. In the later sections (Section 4.2 to 4.6), we give an overview of different priority-queue implementations using bi-

nary heap, binomial heap, Fibonacci heap and strict Fibonacci heap data-structures, and compare their time complexity.

A simple and straight-forward implementation of a priority queue can be done using an ordered array by storing elements in the decreasing order of their priority. Using ordered arrays we can perform INSERT, DELETE and DECREASE-KEY priority-queue operations in $O(n)$ time-steps and EXTRACT-MIN operation in $O(1)$ time-steps. Even though the implementation is inherently simple it is inefficient. So an efficient yet simple implementation of a priority queue using binary heap was introduced by Williams [129].

4.2 Binary heaps

A *heap-ordered tree* is a tree subject to a constraint that the priority of the parent vertex is always less than its children. To be precise, if a vertex v is a child of vertex u in a heap-ordered tree then it must satisfy the condition $\mathsf{key}(v) \geq \mathsf{key}(u)$ and this is referred as the *heap condition* by Knuth [81].

A *binary heap* is a heap-ordered complete binary tree satisfying the heap condition. Consequently, the root of a binary heap is always the smallest element. A n-element binary heap can be implemented using a n-element one-dimensional array with no additional space requirements; an element at index $1 \leq i \leq n$ has its left-child at index $2i$ and its right-child at index $2i + 1$. Consequently, the parent of an element at index $i \geq 2$ is located at index $\lfloor i/2 \rfloor$. An example of a binary heap data-structure is shown in Figure 4.1.

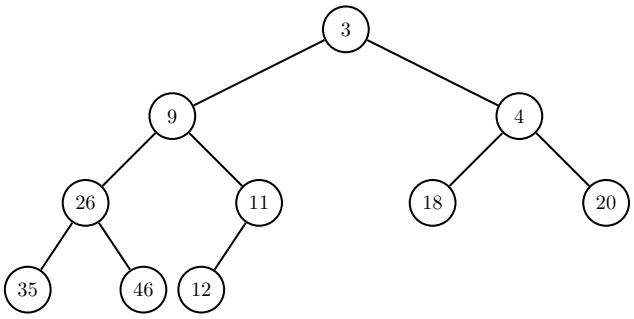


Figure 4.1: An example of a binary heap with 10 elements. The element with minimum priority is the root of the binary heap and the priority of parent vertex is greater than its children. In this example, element 3 is the root vertex and its left and right children are 9 and 4, respectively.

The binary heap data-structure was introduced by Williams [129] in 1964 to improve the complexity of sorting in the *heap-sort* algorithm and it was later used by Floyd [41] in the *tree-sort* algorithm. Using binary heap data-structure INSERT, EXTRACT-MIN, DELETE and DECREASE-KEY priority queue operations can be performed in $O(\log n)$ time-steps, and MERGE priority queue operation in $O(n)$ time-steps.

4.3 Binomial heaps

A *binomial heap* is a forest of heap-ordered binomial trees with unique ranks. A binomial heap can have more than one binomial tree and a non-empty binomial heap of n-elements has at least one and at most $\lceil \log_2 n \rceil$ binomial trees. In addition, the min-heap pointer $\mathsf{min}(H)$ points to the minimum element in the heap. An example of a binomial heap data-structure is shown in Figure 4.2.

The binomial heap data-structure was introduced by Vuillemin [123, 124] in 1977 to speed-up the merge operation in heaps. Using binomial heap DELETE and EXTRACT-MIN operations can be performed in $O(\log n)$ time-steps. Additionally, INSERT and MERGE operations can be performed in $O(1)$ and $O(\log n)$ amortised time-steps, respectively.

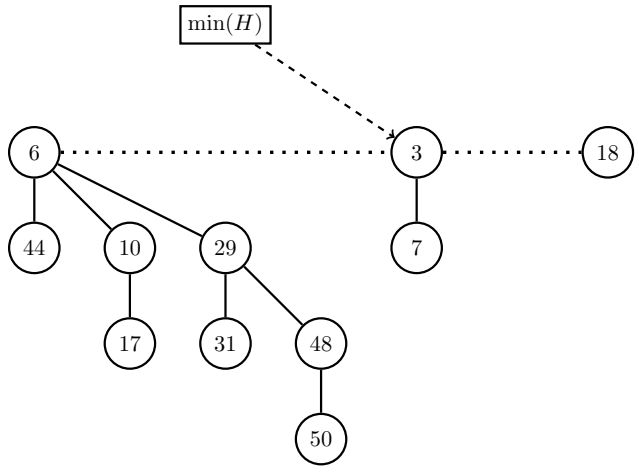


Figure 4.2: An example of a binomial heap with three binomial trees. The min-heap pointer $\mathsf{min}(H)$ points to the minimum element in the heap; in this example 3 is the minimum element and the min-heap pointer is represented by a dotted arrow. In addition, all elements (vertices) in the root-list are connected by dotted lines.

4.4 Fibonacci heaps

A *Fibonacci heap* is a collection of item-disjoint heap-ordered trees. The Fibonacci heap data-structure was introduced by Fredman and Tarjan [47] in 1984 to improve the amortised time complexity of the heap operations which is more realistic when dealing with a sequence of operations than the average or worst-case analysis. Using Fibonacci heaps improved the time complexity of Dijkstra's algorithm to $O(m+n\log n)$ time-steps, furthermore, it improved the best known bounds for the all-pairs shortest path problem, the assignment problem and the minimum spanning tree problem (Cormen *et al.* [24]; Johnson [72]). A Fibonacci heap supports EXTRACT-MIN and DELETE operations in $O(\log n)$ amortised time, DECREASE-KEY operation in $O(1)$ amortised time and all the other heap operations in $O(1)$ constant time.

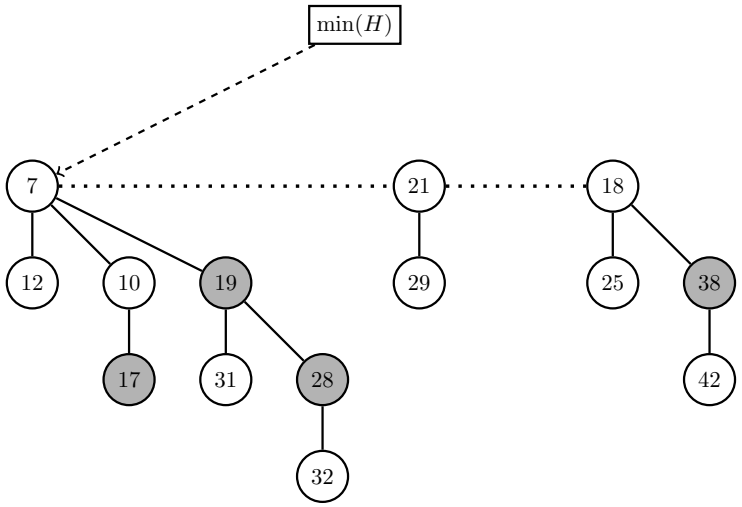


Figure 4.3: An example of a Fibonacci heap data-structure with three Fibonacci trees and root-list $\{7, 21, 18\}$. The pointer $\mathsf{min}(H)$ points to minimum element of the root-list and hence minimum element in the heap; in this example 7 is the minimum element and the min-heap pointer is represented by a dotted arrow. All elements in the root-list are connected by dotted lines. In addition, the shaded elements represent the marked value TRUE.

To achieve the claimed complexity Fredman and Tarjan [47] introduced the following representation of the Fibonacci heap: each element e in heap H contains a pointer $\mathsf{parent}(e)$ which points to its parent element or **nil** if e has no parent, and a pointer $\mathsf{child}(e)$ which points to an arbitrary child of e or **nil** if e has no children. The *child-list* of an element e is the set of children of e and it is denoted by $\mathsf{childlist}(e)$. All the elements in $\mathsf{childlist}(e)$ are doubly linked

in a circular-list. To be precise, each element e has two pointers $\mathsf{left}(e)$ and $\mathsf{right}(e)$ which points to its left and right siblings, respectively, in a circular list or points to itself if e has no siblings. The *root-list* of the heap is the set of root elements and it is denoted by $\mathsf{rootlist}(H)$. All the elements in $\mathsf{rootlist}(H)$ are connected in a circular list. In addition, a min-heap pointer $\mathsf{min}(H)$ points to the element with minimum priority in the root-list and thus the minimum element in the heap. Finally, e also contains a field $\mathsf{rank}(e)$ indicating its rank (rank is the number of children of e) and $\mathsf{marked}(e)$ indicating whether the element is marked. We shall discuss the use of marked field while describing the DECREASE-KEY operation. The *size* of H is the number of elements in H and it is denoted by $\mathsf{n}(H)$. An example of a Fibonacci heap data-structure is given in Figure 4.3 and the aforementioned representation of the Fibonacci heap is shown in Figure 4.4.

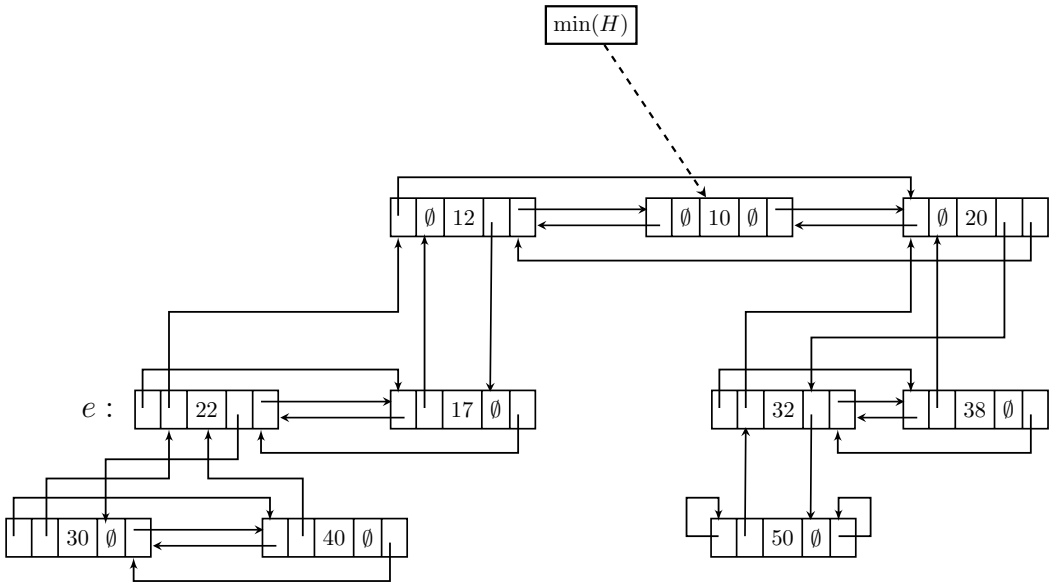


Figure 4.4: A pointer representation of the Fibonacci heap. Each element has four pointers indicating its left sibling, parent, child and right sibling. The pointers are represented by arrows. The middle field indicate the value of the element. The rank and marked fields are not shown in the illustration, and $\emptyset$ symbol indicate **nil**. For example, let e be the element with value 22 then $\mathsf{parent}(e)$ points to its parent element 12, $\mathsf{child}(e)$ points to an arbitrary child 30, $\mathsf{left}(e)$ points to its left sibling 17 and $\mathsf{right}(e)$ points to its right sibling 17. The min-heap pointer $\mathsf{min}(H)$ points to minimum element of the heap and it is represented by a dotted arrow. In this example, 10 is the minimum element.

We perform INSERT, EXTRACT-MIN and DECREASE-KEY priority-queue operations on the example given in Figure 4.3 and display the results in Figure 4.5, Figure 4.6, and Figure 4.7, respectively. In this section, we follow the notations and terminology used by Prömel and Steger [103].

INSERT(H, x): Insert a new vertex x into the root-list of H and update $\mathsf{min}(H)$ if $\mathsf{key}(x)$ is less than $\mathsf{key}(\mathsf{min}(H))$. An illustration of the INSERT operation in Fibonacci heap is shown in Figure 4.5. The pseudo-code of INSERT operation is available in Procedure 1.

Procedure 1: INSERT(H, x)

```
1 rank(x) ← 0
2 child(x) ← nil
3 parent(x) ← nil
4 marked(x) ← FALSE
  // Add x to root list of H
5 rootlist(H) ← rootlist(H) ∪ {x}
6 if key(x) < key(min(H)) then
7 |   update min(H)
8 end
9 n(H) ← n(H) + 1
```

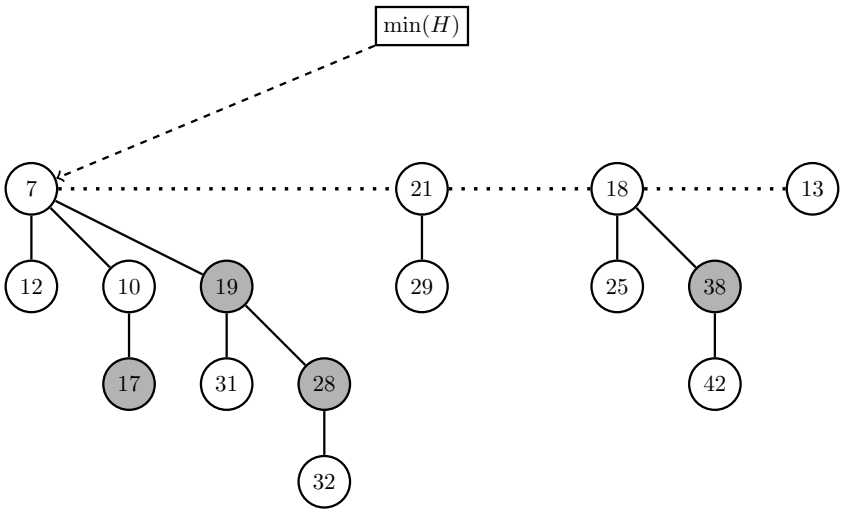


Figure 4.5: An illustration of INSERT operation in Fibonacci heap. Fibonacci heap after inserting a new element 13.

EXTRACT-MIN(H): Remove and return the vertex with minimum priority and reorganise the heap such that no two root vertices have the same rank. The process of reorganising the heap is known as *consolidation* and it is done by recursively combining the Fibonacci trees with the same rank. More precisely, we combine the trees with the same rank by making root vertex of the tree with high priority as the child of root vertex of tree with low priority. The pseudo-code of CONSOLIDATE and EXTRACT-MIN operations is available in Procedure 2 and Procedure 3, respectively. An example of EXTRACT-MIN operation in Fibonacci heap is illustrated in Figure 4.6.

Procedure 2: CONSOLIDATE(H)

```
1  for i ← 1 to ⌊log_φ n(H)⌋ do
2  |  A(i) ← nil
3  end
4  do
5  |  x ← an arbitrary vertex from rootlist(H)
6  |  rootlist(H) ← rootlist(H) \ {x}
7  |  while A(rank(x)) ≠ nil do
8  |  |  y ← A(rank(x))
9  |  |  A(rank(x)) ← nil
10 |  |  if key(x) > key(y) then
11 |  |  |  exchange x ↔ y
12 |  |  end
   |  |  // make y child of x
13 |  |  childlist(x) ← childlist(x) ∪ {y}
14 |  |  rank(x) ← rank(x) + 1
15 |  end
16 |  A(rank(x)) ← x
17 while rootlist(H) ≠ ∅;
18 min(H) ← nil
19 for i ← 1 to ⌊log_φ n(H)⌋ do
20 |  if A(i) ≠ nil then
21 |  |  rootlist(H) ← rootlist(H) ∪ {A(i)}
22 |  |  if key(A(i)) < key(min(H)) then
23 |  |  |  update min(H)
24 |  |  end
25 |  end
26 end
```

Procedure 3: EXTRACT-MIN(H)

```
1  z ← min(H)
2  if z ≠ nil then
3  |  for x ∈ childlist(z) do
4  |  |  parent(x) ← nil
5  |  |  marked(x) ← False
6  |  |  rootlist(H) ← rootlist(H) ∪ {x}
7  |  end
8  |  rootlist(H) ← rootlist(H) \ {z}
9  |  n(H) ← n(H) − 1
10 |  if rootlist(H) = ∅ then
11 |  |  min(H) ← nil
12 |  else
13 |  |  Consolidate(H)
14 |  end
15 end
16 return z
```

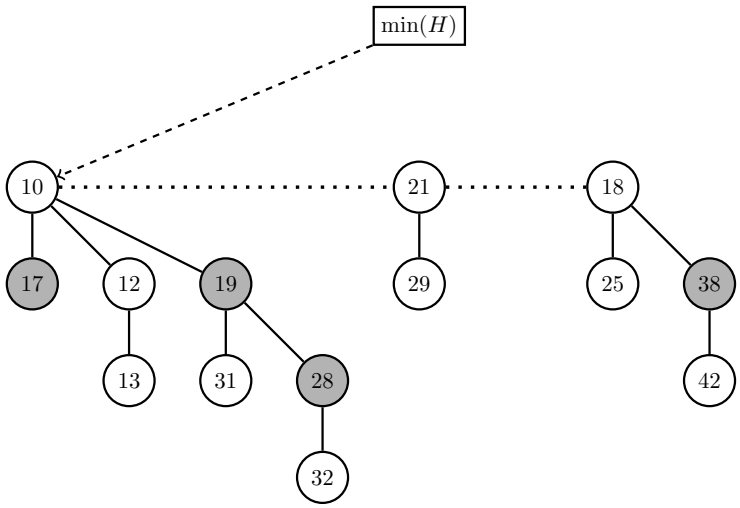


Figure 4.6: An illustration of EXTRACT-MIN operation in Fibonacci heap. Fibonacci heap after extracting the minimum vertex 7.

DECREASE-KEY(H, x, k): Remove the edge between vertex x and its parent vertex $y = \mathsf{parent}(x)$, and insert x with its new priority k in to the root-list. If y has not previously lost a child, to be precise, if $\mathsf{marked}(y) = \text{FALSE}$ then set $\mathsf{marked}(y)$ to TRUE. On the other hand, if y has already lost a child, that is if $\mathsf{marked}(y) = \text{TRUE}$ then remove the edge between y and $\mathsf{parent}(y)$. Fur-

thermore, repeat the procedure for $\mathsf{parent}(y)$ until a vertex is reached which either belongs to the root-list or its marked value is FALSE. Each operation of separating a vertex from its parent is called a *cut-operation* and there is no upper bound for the number of cut-operations. Hence, these operations are also called *cascading-cuts*. The pseudo-code of CUT and DECREASE-KEY operations is available in Procedure 4 and Procedure 5, respectively. An example of DECREASE-KEY operation in Fibonacci heap is illustrated in Figure 4.7.

Procedure 4: CUT(H, x)

```
1  y ← parent(x)
2  parent(x) ← nil
3  marked(x) ← False
4  rank(y) ← rank(y) − 1
5  childlist(y) ← childlist(y) \ {x}
6  rootlist(H) ← rootlist(H) ∪ {x}
7  if key(x) < key(min(H)) then
8  |  update min(H)
9  end
10 if parent(y) ≠ nil then
11 |  if marked(y) = True then
12 |  |  Cut(H, y)
13 |  else
14 |  |  marked(y) ← True
15 |  end
16 end
```

Procedure 5: DECREASE-KEY(H, x, k)

```
1  if k < key(x) then
2  |  error("Invalid key")
3  end
4  key(x) ← k
5  if parent(x) ≠ nil and key(x) < key(parent(x)) then
6  |  Cut(H, x)
7  end
```

Before proceeding to analyse the time complexity of the priority-queue operations in Fibonacci heap, let us recall some basic properties of Fibonacci numbers which are essential for analysing the amortised time complexity of the Fibonacci heap operations.

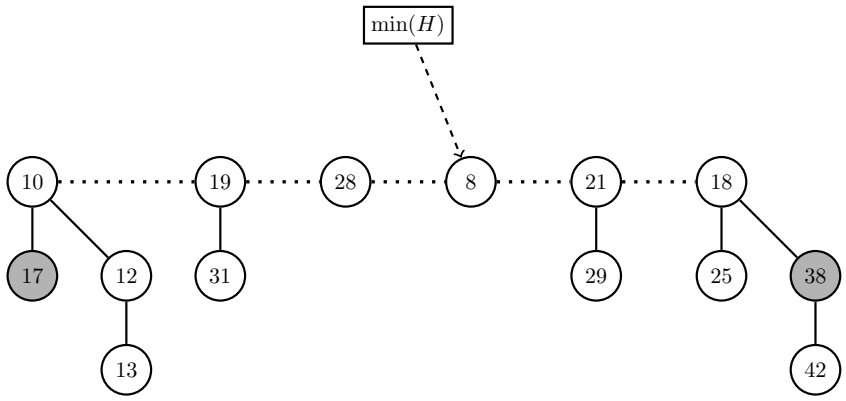


Figure 4.7: An example of DECREASE-KEY operation. Fibonacci heap after decreasing the priority of an element with old priority 32 to new priority 8.

Proposition 4.1. Let F_k be the kth Fibonacci number then $F_k \leq \phi^{k-1}$ for all $k \in \mathbb{Z}_+$ where $\phi = (1+\sqrt{5})/2$ is the golden ratio.

Proof. Let $F_0, F_1, F_2, \ldots$ be the Fibonacci sequence. By the rule $F_0 = 0$, $F_1 = 1$ and every further term is the sum of the preceding two. Hence $F_k = F_{k-1} + F_{k-2}$ for all $k \geq 2$.

For $k = 1$, $F_1 = 0 \leq \phi^0 = 1$ and for $k = 2$, $F_2 = 1 \leq \phi^{2-1} = \phi^1$, this establishes the base case. Let us assume that the formula holds for all integers less than k then $F_{k-1} \leq \phi^{k-2}$ and $F_{k-2} \leq \phi^{k-3}$. By adding these inequalities we get,

$$F_k = F_{k-1} + F_{k-2} \leq \phi^{k-2} + \phi^{k-3} = \phi^{k-3}(\phi + 1). \tag{4.1}$$

The golden ratio satisfies,

$$1 + \phi = \phi^2. \tag{4.2}$$

By substituting (4.2) in (4.1) we get $F_k \leq \phi^{k-1}$. □

Remark. Similarly, we can also prove that $F_k \geq \phi^{k-2}$ and furthermore $\phi^{k-2} \leq F_k \leq \phi^{k-1}$.

Lemma 4.2. Let u be any vertex in a Fibonacci heap such that $\mathsf{rank}(u) = k$ and v_i be the ith child of u in the order they are linked to u from oldest to newest then the rank of v_i is greater than or equal to $i-2$ for all $i = 1, \ldots, k$.

Proof. Let vertex v_i be the ith child of vertex u then before linking v_i to u, u had at least $i-1$ children. Hence $\mathrm{rank}(u) \geq i-1$. If v_i is linking to u this means that v_i and u have the same rank before linking. So $\mathsf{rank}(u) =$

$\mathsf{rank}(v_i) \geq i - 1$. However, after linking v_i to u, the rank of v_i could have decreased by at most one without causing v_i to be cut as a child of u. Hence $\mathsf{rank}(v_i) \geq i - 2$. □

Lemma 4.3. In a Fibonacci heap a subtree rooted at a vertex with rank k has at least F_{k+2} descendants and $F_{k+2} \geq \phi^k$ where F_k is the kth Fibonacci number and $\phi = (1 + \sqrt{5})/2$.

Proof. Let S_k denote the minimum number of descendants of a vertex with rank k. Clearly, $S_0 = 1$, $S_1 = 2$ and from Lemma 4.2 we have that $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ for all $k \geq 2$. Since the Fibonacci numbers satisfy the relation $F_{k+2} = \sum_{i=0}^{k} F_i + 2$ for all $k \geq 2$ we have $S_k \geq F_{k+2}$ for all $k \geq 0$ by induction on k. Furthermore, from Proposition 4.1 we have that $F_{k+2} \geq \phi^k$. Hence there are at least ϕ^k descendants rooted at a vertex with rank k. □

Remark. Lemma 4.3 is the source of the name *Fibonacci heap*.

Corollary 4.4. Let H be a Fibonacci heap with at most n elements then rank of each vertex in $u \in H$ is at most $\log_\phi n$ where $\phi = (1 + \sqrt{5})/2$.

Proof. From the given data we can infer that any subtree rooted at u can contain at most n vertices. In addition, from Lemma 4.3 we obtain $\phi^{\mathrm{rank}(u)} \leq n$, by taking log on both sides we get $\mathrm{rank}(u) \leq \log_\phi n$. □

4.5 Amortised analysis

Amortisation or *averaging over time* is a powerful technique used for the complexity analysis of data-structures. The theory of algorithms has traditionally focused on worst-case analysis and this focus has led to many efficient algorithms. Even though the worst-case analysis is a good complexity measure, there are a number of problems for which it does not provide empirically accurate results. On the other hand, amortised analysis reports more robust and practical runtimes. Most importantly we can obtain tight upper and lower bounds on a variety of algorithms. There are two well-known approaches used for amortised analysis: (i) the *banker's view*, which was implicitly used by Brown and Tarjan [17] for analysing the complexity of 2-3 trees and it was developed more fully by Huddleston and Mehlhorn [64, 65] for analysing the complexity of generalised B-trees; and (ii) the *physicist's view*, which was introduced by Sleator and Tarjan [112] for analysing the complexity of the paging rules in self-organising lists. We use the latter approach to analyse the amortised time complexity of the Fibonacci heap.

In physicist's view, Sleator and Tarjan [112] define a *potential function* Φ that maps any configuration of the data-structure to a rational number called the *potential*. The *amortised time* a_i of the ith operation is given by $t_i + \Phi_i - \Phi_{i-1}$, where t_i denotes the actual time, Φ_i denotes the potential after ith operation and Φ_{i-1} denotes the potential after $i-1$th operation. Φ_0 denotes the potential before the first operation. Using this notation any sequence of m operations satisfies the following equality,

$$\sum_{i=1}^{m} t_i = \sum_{i-1}^{m} a_i - \Phi_i + \Phi_{i-1} = \Phi_0 - \Phi_m + \sum_{i=1}^{m} a_i.$$

Another approach for amortised complexity analysis is the banker's view which is beyond the scope of this thesis. For a detailed explanation of amortised analysis we refer the reader to a survey of amortised computational complexity by Tarjan [114] and an introductory book of algorithms by Cormen *et al.* [24, Chapter 17].

Lemma 4.5 (Fredman and Tarjan [47])**.** In a Fibonacci heap starting with an empty heap any arbitrary sequence of k operations of INSERT, DECREASE-KEY and EXTRACT-MIN can be performed in $O(k + l \log n)$ time-steps, where l is the number of EXTRACT-MIN operations and n is the maximum number of elements contained in the heap at any time.

Proof. Let r_i denote the number of vertices in the root-list and m_i denote the number of marked vertices after i operations. Let T_i denote the time spent during first i operations and C be a fixed sufficiently large integer. The potential function is defined as,

$$\Phi_i = C(r_i + m_i). \tag{4.3}$$

The amortised runtime of the ith operation is given by,

$$a_i = (\Phi_i + T_i) - (\Phi_{i-1} + T_{i-1}). \tag{4.4}$$

It is sufficient to show that a_i is constant whenever ith operation is an INSERT or a DECREASE-KEY operation, and it is bounded by $O(\log n)$ for an EXTRACT-MIN operation. To prove this we consider the following three cases:

1. If the ith operation is INSERT then the number of root vertices is $r_i = r_{i-1} + 1$ and the number of marked vertices is $m_i = m_{i-1}$. The time spent during first i operations is $T_i \leq T_{i-1} + C$. By substituting these inequalities in (4.4) we get $a_i \leq 2C$.

2. If the ith operation is EXTRACT-MIN then the number of root vertices is $r_i \leq \log n$ and the number of marked vertices is $m_i \leq m_{i-1}$. From Corollary 4.4 the vertex with the minimum key has at most $\log n$ children. In addition, we observe that each linking of two vertices in the root list reduces the number of vertices in the root list by one. Hence, $T_i \leq T_{i-1} + C(r_{i-1} + \log n)$. By substituting these inequalities in (4.4) we get $a_i \leq 2C \log n$.

3. If the ith operation is DECREASE-KEY then let us assume that x be the number of calls made for Procedure CUT. Hence, the number of root vertices is $r_i = r_{i-1} + x$, the number of marked vertices is $m_i \leq m_{i-1} - (x-1)+1$ and the time spent during first i operations is $T_i \leq T_{i-1} + Cx$. By substituting these inequalities in (4.4) we get $a_i \leq 4C$. □

4.6 Strict Fibonacci heaps

A pointer based implementation of the priority queue with time bounds matching those of a Fibonacci heap in the worst-case was introduced by Brodal, Logogiannis and Tarjan [14] in 2012, known as the *strict Fibonacci heap*. Using strict Fibonacci heap INSERT, CONSOLIDATE and DECREASE-KEY operations can be done in worst-case $O(1)$ time furthermore, EXTRACT-MIN and DELETE operations in worst-case $O(\log n)$ time, where n is the size of the heap.

However, for the purpose of this thesis we restrict our discussion to Fibonacci heap, which is sufficient to achieve the claimed time complexity of $O(m + n \log n)$ for computing a shortest path in graphs using Dijkstra's algorithm.

Table 4.1 lists the time complexity of the priority-queue operations using ordered array, binary, binomial, Fibonacci and strict Fibonacci heap data-structures.

4.7 The shortest path problem

The *shortest path problem* is one of the fundamental problems in computer science and it arises as a sub-problem while solving many optimisation problems such as the all-pairs shortest path problem, the spanning tree problem, the Steiner problem and many more. It is important to note that the short-

Operation	ordered array	binary heap	binomial heap	Fibonacci heap	strict Fibonacci
INSERT	$O(n)$	$O(\log n)$	$O(1)^*$	$O(1)$	$O(1)$
DELETE	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$	$O(\log n)$
EXTRACT-MIN	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$	$O(\log n)$
DECREASE-KEY	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)^*$	$O(1)$
MERGE	$-$	$O(n)$	$O(\log n)^*$	$O(1)$	$O(1)$

Table 4.1: The time complexity of the priority queue operations using ordered array, binary, binomial, Fibonacci and strict Fibonacci heap data-structures. The amortised complexity is denoted by $O(\cdot)^*$ and the worst-case complexity is denoted by $O(\cdot)$.

est path problem is also referred as the *single-pair shortest path problem* to distinguish from the *single-source shortest path problem* (Knuth [80]). The shortest path problem plays an important role in studying the Steiner and group Steiner problems in graphs, and we shall see this in Chapter 5.

The shortest path problem. Given a network $N = (V, E, w)$ and two vertices $u, v \in V$, the shortest path problem is to find a path between u and v in N such that the length of path is minimised.

The single-source shortest path problem. Given a network $N = (V, E, w)$ and a source vertex $s \in V$, the single-source shortest path problem is to find paths from s to every vertex $v \in V \setminus \{s\}$ such that the length of each path is minimised.

Our historical survey of the algorithms for solving the Shortest path problem in graphs is based on the survey by Schrijver [109]. The matrix method for graphs with unit edge weight was studied by Landhal and Runge [87]. In the matrix method, a directed graph is represented using a matrix and the distance between the vertices is calculated by iterative matrix product and transitive closure property of the graph. The matrix method was later studied by Landahl [86]; Luce and Perry [91]; and Luce [90]. An algorithm based on the matrix method for solving the shortest path problem in graphs with unit length was presented by Shimbel [110] in 1954 in the context of finding the all-pairs shortest path with $O(n^4)$ time-steps. Later, Ford [46] in 1956 presented an algorithm with $O(n^2m)$ time complexity and it was improved to $O(nm)$ by Bellman [7] in 1958. Dantzig in 1958 gave a $O(n^2 \log n)$ algorithm for graphs with nonnegative weights associated with the edges. Dijkstra [31] in 1959 gave a concise description of the *Dijkstra's method* yielding a $O(n^2)$ time implementation. However, using more efficient data-structures

such as the binomial [73] and Fibonacci [47] heaps, Dijkstra's method can be implemented with running time $O(m \log n)$ and $O(m + n \log n)$, respectively. Karlsson and Poblete [76] in 1982 presented a $O(m \log \log C)$ time algorithm for the shortest path problem, where C is the maximum edge weight in the graph. Gabow [51] in 1985 improved this to $O(m \log C)$. In 1990, Ahuja *et al.* [1] presented a $O(m+n\sqrt{\log C})$ time algorithm. Thorup [116] presented an algorithm with $O(m)$ time-steps for graphs with nonnegative integer weights associated with the edges in 1999. In 2004, Thorup [117] presented an algorithm with $O(m+n \log \log n)$ complexity using a Fibonacci heap style integer priority queue.

4.8 Dijkstra's Algorithm

In this section, we present an algorithm based on the visit-and-label approach introduced by Dijkstra to solve the shortest path problem for undirected graphs with nonnegative integer weights associated with the edges. The original variant of Dijkstra's algorithm finds a shortest path between two vertices in a graph. However, a more-common variant of this algorithm can find shortest paths from a single-source vertex to all other vertices in the graph. The pseudo-code of the more-common variant of Dijkstra's algorithm is available in Algorithm 1.

Let $N = (V, E, w)$ be a connected network with nonnegative integer weights associated with the edges and $s \in V$ be the source vertex. The algorithm partitions V into two subsets: (i) a subset $Q \subseteq V$ of queued vertices (to be visited later); and (ii) a subset $V \setminus Q$ of visited vertices. For all $x \in V$, let $d(x)$ denotes the length of a shortest s-x path P computed by the algorithm and $p(x)$ denotes the vertex through which x is connected to s along P. At each iteration of the while loop (Lines 11–21 in Algorithm 1) a vertex $u \in Q$ which satisfies $d(u) = \min\{d(z) : \text{ for all } z \in Q\}$ is removed from Q and added to $V \setminus Q$. Furthermore, for each vertex $v \in \mathsf{adj}(u)$ such that $v \in Q \setminus \{u\}$, $d(v)$ and $p(v)$ are updated if $d(v) > d(u) + w(\{u, v\})$ by relaxing the edges incident to u.

Lemma 4.6. Dijkstra's algorithm is correct and computes the shortest path in $O(m + n \log n)$ time-steps.

Proof. To prove the correctness of Dijkstra's algorithm we show that the following statements hold at the end of each iteration of the while loop (Lines 11–21 in Algorithm 1):

1. For all $x \in Q$ the length of a shortest s-x path in the network N' induced by the vertex set $(V \setminus Q) \cup \{x\}$ is equal to $d(x)$ and there exists such a shortest s-x path in N' ending with the edge $\{p(x), x\}$ such that $p(x) \in V \setminus Q$, and

2. For all $x \in V \setminus Q$ the length of a shortest s-x path is equal to $d(x)$ and there exists an s-x path which ends with the edge $\{p(x), x\}$ such that $p(x) \in V \setminus Q$ if $u \neq s$ and $p(x) = \textbf{nil}$ otherwise.

As base case, at the end of the first iteration note that $Q = V \setminus \{s\}$, $V \setminus Q = \{s\}$ and $u = s$. The distance $d(s) = 0$ and $p(s) = \textbf{nil}$, and for all $v \in \mathsf{adj}(s)$, $d(v) = w(\{s, v\})$ and the shortest $s - v$ path ends with the edge $\{s, v\}$ such that $p(v) = s$. Hence, the statements 1 and 2 hold at the end of the first iteration. Thus to verify the correctness of the algorithm, it is sufficient to show that if the statements 1 and 2 hold at the beginning of an iteration of the while loop then they also hold at the end with respect to the visited set.

By statement 1 of the assumption, there exists an s-u path P in network N' induced by the vertex set $(V \setminus Q) \cup \{u\}$ ending with the edge $\{p(u), u\}$ such that $W(P) = d(u)$. We argue that $d(u)$ is equal to minimum. For the sake of contradiction choose an arbitrary s-u path P' that is not totally contained in N' such that $W(P') < W(P)$. This is possible only if P' contains at least one intermediate vertex $j \notin V \setminus Q$. From the choice of u it still holds that $W(P) = d(u) \leq d(j) \leq W(P')$, however this contradicts our assumption that $W(P') < W(P)$. Furthermore, for all $x \in V \setminus Q$, $d(x)$ is equal to minimum still holds. Hence, statement 2 holds for $(V \setminus Q) \cup \{u\}$. From the structure of the algorithm it is clear that: for all $v \in \mathsf{adj}(u)$ such that $v \in Q \setminus \{u\}$ we update $d(v)$ and $p(v)$ strictly if $d(v) > d(u) + w(\{u, v\})$ (see Lines 14–20 in Algorithm 1). Hence, statement 1 holds for $V \setminus Q \cup \{u\}$.

Let us analyse the time complexity of Dijkstra's algorithm. Clearly, the initialisation phase takes $O(n)$ time-steps and throughout the visit-and-label phase we execute at most $n - 2$ iterations of Extract-min operation and at most m iterations of Decrease-key operation. The complexity of the algorithm mainly depends on extracting the minimum distance vertex and updating the distances. Using Fibonacci heap, we can perform Extract-min and Decrease-key operations in $O(\log n)$ and $O(1)$ amortised time, respectively (see Lemma 4.5). Furthermore, labelling of all the neighbours can certainly be done in $2m$ time-steps as each edge is considered at most twice. Hence, Dijkstra's algorithm has a time complexity of $O(m + n \log n)$. □

Algorithm 1: DIJKSTRA

Input: A network $N = (V, E, w)$, $w : E \to \mathbb{Z}_{\geq 0}$ and a source vertex s.
Output: d, minimum distance from s to $v \in V \setminus \{s\}$
p, previous vertex in the path from s to $v \in V \setminus \{s\}$.

```
// Initialisation
```
$d(s) \leftarrow 0$
$p(s) \leftarrow$ **nil**
visit$(s) \leftarrow$ TRUE
INSERT(Q, s)
for all $v \in V \setminus \{s\}$ **do**
  $d(v) \leftarrow \infty$
  $p(v) \leftarrow$ **nil**
  visit$(v) \leftarrow$ FALSE
  INSERT(Q, v)
end
```
// Visit and label
```
while n$(Q) > 0$ **do**
  $u \leftarrow$ EXTRACT-MIN(Q)
  visit$(u) \leftarrow$ TRUE
  for all v adjacent to u **do**
    if $d(u) + w(\{u, v\}) < d(v)$ and visit$(v) =$ FALSE **then**
      $d(v) \leftarrow d(u) + w(\{u, v\})$
      $p(v) \leftarrow u$
      DECREASE-KEY$(Q, v, d(v))$
    end
  end
end
return d, p

Chapter 5

Algorithms for the Steiner problem in graphs

In this chapter, we give a survey of the existing work and summarise the algorithms presented to date for the Steiner problem and the group Steiner problem. We discuss two parameterised algorithms for solving the Steiner problem and a transformation to solve the group Steiner problem as the Steiner problem. We begin this chapter with a review of approximation and exact algorithms for the Steiner problem in Section 5.1 and Section 5.2, respectively. In Section 5.3, we discuss an algorithm based on dynamic programming presented by Dreyfus and Wagner [34]. In Section 5.4, we discuss a procedure for constructing a Steiner tree using the bookkeeping information stored in the Dreyfus–Wagner algorithm. In Section 5.5, we discuss an improvement of the Dreyfus–Wagner algorithm presented by Erickson, Monma and Veinott [38]. It is a parameterised algorithm which runs in edge-linear time and the exponential part can be restricted to the number of terminals. Finally, we discuss a linear-time reduction presented by Voß [122] for solving the group Steiner problem as the Steiner problem in graphs in Section 5.6.

Let us recall that the Steiner problem in graphs is $\mathcal{NP}$-complete. In practice there exists no efficient algorithm to find the optimal solution for most large-scale Steiner problems. On the other hand, for some applications it is sufficient even if the obtained solution is not optimal and the concept of approximation algorithms is introduced for this purpose.

5.1 Approximation algorithms

An *optimisation problem* Π is either a *minimisation problem* or a *maximisation problem*. Each valid *input instance* I of Π comes with a non-empty set of *feasible solutions* and each feasible solution is assigned a nonnegative rational number called its *objective function value*. A feasible solution which achieves the optimal objective function value for I is called the *optimal solution* of I and it is denoted as $OPT(I)$. In this thesis, we only consider the instances with nonnegative integer objective function value.

An *approximation algorithm* is used to obtain a sub-optimal solution in polynomial time and it is usually associated with $\mathcal{NP}$-hard problems. A *sub-optimal* solution means that the solution is not too far from the optimal solution or it is within a guaranteed factor of the optimal solution. For a detailed discussion of concepts and terminology related to approximation algorithms, we refer the reader to a book by Vazirani [120]. In what follows, n denotes the number of vertices, m denotes the number of edges and k denotes the number of terminals in the input graph instance.

A simple 2-approximation algorithm for the Steiner problem based on minimum spanning tree heuristic was presented by Gilbert and Pollak [56] in 1968. For more than twenty-five years no better approximation algorithm for the problem was known. In 1993, Zelikovsky [130] presented a greedy algorithm based on 3-Steiner trees with an approximation ratio of 1.834. The approach was extended by Berman and Ramaiyer [8] in 1994 using k-Steiner trees and resulted in an improved approximation ratio of 1.694. Karpinski and Zelikovsky [79] in 1997 obtained an approximation ratio of 1.644 using a novel technique of choosing the Steiner points in dependence on the possible deviation from the optimal solution that minimises the weighted sum of the length. Hougardy and Prömel [62] used a similar approach resulting in an approximation algorithm that is at most a factor of 1.598 away from the optimal solution. Robins and Zeliskovsky [105] in 2005 presented a greedy algorithm with approximation ratio of 1.550. Bykra *et al.* [18] improved the approximation factor to $\ln(4)+\epsilon < 1.39$ by developing an LP-based algorithm. Furthermore, Chlebík and Chlebíková [21] gave inapproximability results and showed that the problem is $\mathcal{NP}$-hard to approximate within a factor 96/95.

To the best of the authors knowledge, the group Steiner problem in graphs was formally introduced by Riech and Widmeyer [104] in 1989. They modelled an interesting application with regard to the layout of integrated circuits into the group Steiner problem and further formulated two heuristics for the problem with an approximation ratio of $k-1$. Dror, Haouari and Chaouachi [36]

in 2000 formulated an agricultural application and tested several heuristics; among them a genetic algorithm performed the best in terms of the solution quality. Garg, Konjevod and Ravi [54] in 1998 gave a poly-logarithmic approximation algorithm and showed that the group Steiner problem is approximable within $O(\log^3 n \log k)$. In 2009 Demaine, Hajiaghayi and Klein [27, 28] showed that the problem is approximable within $O(\log^2 n)$ if the graph is a tree. Bateman *et al.* [6] have given an algorithm with sub-linear performance guarantee with an approximation ratio of $(1 + \ln n/2)\sqrt{k}$, this ratio comes by approximating the group Steiner tree by a 2-star and then approximating the 2-star within a logarithmic factor. Halperin and Krauthgamer [58] gave inapproximability results and showed that the group Steiner problem is not approximable within $\Omega(\log^{2-\epsilon} n)$ unless $\mathcal{NP}$ admits quasi-polynomial time Las Vegas algorithm.

The scope of this thesis is to present exact algorithms for the Steiner and group Steiner problems. We mainly concentrate on scalable algorithms, in particular, parameterised algorithms for the problem. Likewise, we do not discuss approximation algorithms in greater detail. In the next section, we give a brief review of the exact algorithms presented for solving the Steiner and group Steiner problems.

5.2 Exact algorithms

An *exact algorithm* for the optimisation problem always finds an optimal solution. Meanwhile, there exists no efficient algorithm to find an optimal solution for a $\mathcal{NP}$-complete problem, unless $\mathcal{P} = \mathcal{NP}$. Giving the $\mathcal{NP}$-completeness proof for the Steiner and group Steiner problems is sufficient to say that there exists no polynomial-time algorithm to solve these problems. All known exact algorithms for the Steiner and group Steiner problems have exponential-time complexity.

An algorithm based on dynamic programming was presented by Dreyfus and Wagner [34] in 1971 with time complexity $O(3^k n + 2^k n^2 + n^3)$. However, the time complexity of the Dreyfus–Wagner algorithm can be improved to $O(3^k n + 2^k n^2 + n(n \log n + m))$ using Dijkstra's algorithm with Fibonacci heap for computing shortest paths. Erickson, Monma and Veinott [38] improved the time complexity of the algorithm by computing the distances more cleverly on demand to $O(3^k n + 2^k(n \log n + m))$ time-steps. We will discuss these two algorithms in more detail in the later sections of this chapter.

As a kind of exact algorithm Downey and Fellows [33] in 1990 investigated a parameterised algorithm for the Steiner problem and showed that the problem is *fixed parameter tractable (FPT)*, in the sense that it can be solved in time $O(f(k)g(n))$ where f is an exponential function, g is a polynomial function and typically k is much smaller than n. A problem allowing such a fixed parameter tractable algorithm is called *fixed parameter tractable problem* (Cygan *et al.* [26]). A fixed parameter tractable algorithm for the Steiner problem is to find a solution in polynomial time with respect to the size m of the host graph (or order n of the host graph) and in exponential time with respect to the number of terminals k. Then, the algorithm for the Steiner problem is in time $f(k)$ times the polynomial of m (or n) and it is denoted as $O^*(f(k))$. Many researchers and computer scientists have tried to reduce the exponential factor in algorithms for the Steiner problem. Mölle, Richter and Rossmanith [96] in 2006 improved the exponent to $(2+\delta)^k \cdot poly(n)$ for arbitrary but fixed $\delta > 0$, Fuchs, Kern and Wang *et al.* [49] in 2007 improved to $O^*(2.684^k)$, futhermore to $O^*(2.38^k)$ [48]. In 2007, Björklund *et al.* [10] improved the time complexity to $O(2^k n^2 + nm)$ using subset convolution and Möbius inversion for graphs with bounded integer edge weights. However, all the algorithms listed above require exponential space.

Fomin, Grandoni and Kratsch [43] in 2008 presented the first polynomial-space algorithm for the Steiner problem with $O(6^k n^{O(\log k)})$ time complexity for edge weighted variant, combining this method with Dreyfus and Wagner for $k < \log n$ Fomin *et al* [44] obtained a $O^*(2^{O(k \log k)})$ time polynomial-space algorithm. Nerderlof [97] in 2009 improved the complexity to $O^*(2^k)$ for edge weighted variant with bounded edge weights using Möbius inversion. Vygen [125] in 2011 presented a polynomial-space algorithm with time complexity in the order of $O(nk2^{k+\log_2 k \log_2 n})$ time-steps. In 2015, Fomin *et al.* [45] presented the first single-exponential time polynomial-space FPT algorithm with time complexity $O(7.97^k n^4 \log W)$ using $O(n^3 \log k \log nW)$ space for graphs with nonnegative edge weights in range 0 to W.

A transformation for solving the group Steiner problem as the Steiner problem in graphs was presented by Voß [122] in 1990, using this transformation most algorithms for the Steiner problem can be used to solve the group Steiner problem. The transformation was restated and a practical implementation was presented by Duin, Volgenant and Voß [37] in 2004.

In the next sections, we present two parameterised algorithms for solving the Steiner problem in graphs: (i) a dynamic-programming algorithm introduced by Dreyfus and Wagner [34]; and (ii) an improvement of the Dreyfus–Wagner algorithm presented by Erickson, Monma and Veinott [38]. Addi-

tionally, we discuss a procedure to extract a Steiner tree from the additional bookkeeping information stored while computing the cost of a Steiner tree in the Dreyfus–Wagner algorithm.

5.3 Dreyfus–Wagner algorithm

Let us recall the Steiner Problem (SP), given a connected network $N = (V, E, w)$ and a subset K of vertices V called terminals. The task is to find a minimum-weight tree T in N that connects all the terminals in K.

For a subset $X \subseteq K$ and a vertex $v \in V$, let $f_v(X)$ denote the length of a Steiner tree spanning vertices in $X \cup \{v\}$, and let $g_v(X)$ denote the length of a Steiner tree spanning vertices in $X \cup \{v\}$ where v has degree at least 2. The length of the tree T is denoted by $W(T) = \sum_{e \in E(T)} w(e)$ and length of a shortest path between vertices u, v is denoted by $d(u, v)$. The pseudo-code of the Dreyfus–Wagner algorithm is available in Algorithm 2.

The Dreyfus–Wagner algorithm exploits the optimal-decomposition property. Given a Steiner tree spanning $X \cup \{v\}$ in which the degree of v is at least two then we can split the tree at v to obtain two subtrees: one spanning $X' \cup \{v\}$; and the other spanning $X \setminus X' \cup \{v\}$ for some non-empty subset $X' \subset X$. Thus we obtain the following recurrence to compute $g_v(X)$,

$$g_v(X) = \min_{\emptyset \neq X' \subset X} \{f_v(X') + f_v(X \backslash X')\}.$$

Given a Steiner tree connecting vertices in $X \cup \{v\}$ if v has degree at least two then a Steiner tree has length $g_v(X)$; otherwise the tree path from vertex v to some vertex $u \in X$ or $u \in V \setminus X$ such that the degree of u is at least three, whichever comes first. Thus we obtain the following recurrence to compute $f_v(X)$,

$$f_v(X) = \min\{\min_{u \in X}\{d(v, u) + f_u(X \setminus \{u\})\}, \min_{u \in V \setminus X}\{d(v, u) + g_u(X)\}\}.$$

The initial conditions are $g_v(\emptyset) = 0$, $f_v(\emptyset) = 0$ and $f_v(\{u\}) = d(v, u)$ for all $u, v \in V$. The problem can be solved recursively in $k - 1$ steps where $k = |K|$ is the number of terminals. Let us prove the validity of the optimal-decomposition property.

Lemma 5.1 (Optimal decomposition [34])**.** Let $X \subseteq K$, $|X| > 1$ and $v \in V$. Then

$$g_v(X) = \min_{\emptyset \neq X' \subset X} \{f_v(X') + f_v(X \backslash X')\} \tag{5.1}$$

and

$$\begin{aligned} f_v(X) = \min\{ &\min_{u \in X}\{d(v,u) + f_u(X \setminus \{u\})\}, \\ &\min_{u \in V \setminus X}\{d(v,u) + g_u(X)\}\}. \end{aligned} \tag{5.2}$$

Proof. To prove the correctness of (5.1), let us assume that T be a Steiner tree connecting vertices in $X \cup \{v\}$ with length $W(T)$ such that $\mathsf{deg}_T(v) \geq 2$. Clearly X can be decomposed into two subsets X' and $X \setminus X'$ for some $\emptyset \neq X' \subset X$ such that X' and $X \setminus X'$ are connected through v. The decomposition of X to X' and $X \setminus X'$ is illustrated in Figure 5.1. Let $T_{X'}$ be a Steiner tree connecting $X' \cup \{v\}$ and $T_{X \setminus X'}$ be a Steiner tree connecting $X \setminus X' \cup \{v\}$ then the weight of T can be decomposed as $W(T) = W(T_{X'}) + W(T_{X \setminus X'})$.

For the sake of contradiction, let us assume that there exists a Steiner tree $T'_{X'} \neq T_{X'}$ connecting all the vertices in $X' \cup \{v\}$ such that $W(T'_{X'}) < W(T_{X'})$ then $T_{X'}$ can be replaced by $T'_{X'}$ in T to get T' that connects vertices in $X \cup \{v\}$. Indeed,

$$W(T') = W(T'_{X'}) + W(T_{X \setminus X'}) < W(T_{X'}) + W(T_{X \setminus X'}) = W(T).$$

However, this contradicts our hypothesis that T is a Steiner tree connecting the vertices in $X \cup \{v\}$. By minimising over all proper subsets $\emptyset \neq X' \subset X$ we get the recursion (5.1).

Let us prove the correctness of recursion (5.2). If $|X \cup \{v\}| = 2$ then $f_v(X)$ is the length of a shortest path between v and $u \in X$. Let us assume that T be a Steiner tree for $X \cup \{v\}$ with $\mathsf{deg}_T(v) \geq 2$ then $f_v(X) = g_v(X)$. This can be obtained by replacing $u = v$ in (5.1).

Let v be a leaf vertex in T, that is $\mathsf{deg}_T(v) = 1$ and P be a shortest path in T connecting v to $u \in V(T)$ and length of path $W(P) = d(v,u)$. Let $T_{X \setminus \{v\}}$ be a Steiner tree connecting all vertices in $X \setminus \{v\}$. Clearly the weight of T can be decomposed as $W(T) = d(v,u) + W(T_{X \setminus \{v\}})$. We distinguish the decomposition of $X \cup \{v\}$ into two different cases: Case (a) vertex $u \in X$; and Case (b) vertex $u \in V \setminus X$. These cases are enumerated in Figure 5.2. By minimising over these two cases we get the recursion (5.2).

Case (a). Let v be a leaf vertex and P be a shortest path from v to $u \in X$ in T and $\mathsf{deg}_T(u) \geq 2$ then T is the union of a Steiner tree for X and

a shortest path from v to u in T. Hence, $f_v(X) = d(v,u) + f_u(X \setminus \{u\})$. As a contradiction, let us assume that there exists a path $P' \neq P$ in T such that $W(P') < W(P)$, then P can be replaced with P' in T to get Steiner tree T' connecting nodes in $X \cup \{v\}$ such that $W(T') < W(T)$. However, this contradicts the hypothesis that T is a Steiner tree connecting the nodes in $X \cup \{v\}$.

Case (b). Let v be a leaf vertex and P be a shortest path in T from v to $u \in V \setminus X$ and $\mathsf{deg}_T(u) \geq 2$ then T is the union of a Steiner tree for X and a shortest path from v to u. Hence, $f_v(X) = d(v,u) + g_u(X)$.

For the sake of contradiction, let us assume that there exists a Steiner tree $T'_{X\setminus\{v\}}$ connecting all vertices in $X \setminus \{v\}$ such that $T'_{X\setminus\{v\}} < T_{X\setminus\{v\}}$. Then, $T_{X\setminus\{v\}}$ can be replaced by $T'_{X\setminus\{v\}}$ in T to obtain Steiner tree T' connecting $X \cup \{v\}$ such that $W(T') < W(T)$. Indeed,

$$W(T') = d(v,u) + W(T'_{X\setminus\{v\}}) < d(v,u) + W(T_{X\setminus\{v\}}) = W(T).$$

However, this contradicts our assumption that T is a Steiner tree. □

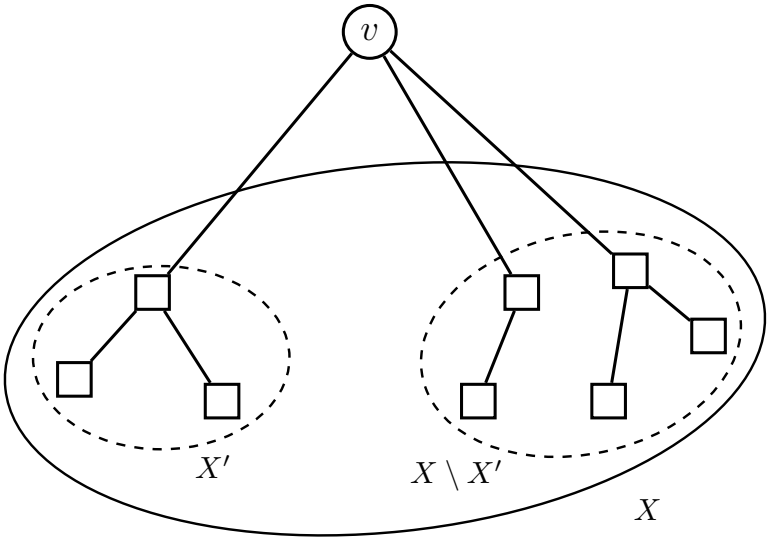


Figure 5.1: An illustration of the optimal-decomposition property for recursion (5.1); terminal vertices are denoted by rectangles and non-terminal vertices are denoted by circles. The set X is decomposed into two proper subsets X' and $X \setminus X'$ which are enclosed inside two-separate dotted regions. The subsets X and $X \setminus X'$ are connected through a vertex $v \in V$ such that degree of vertex v is $\mathsf{deg}_T(v) \geq 2$.

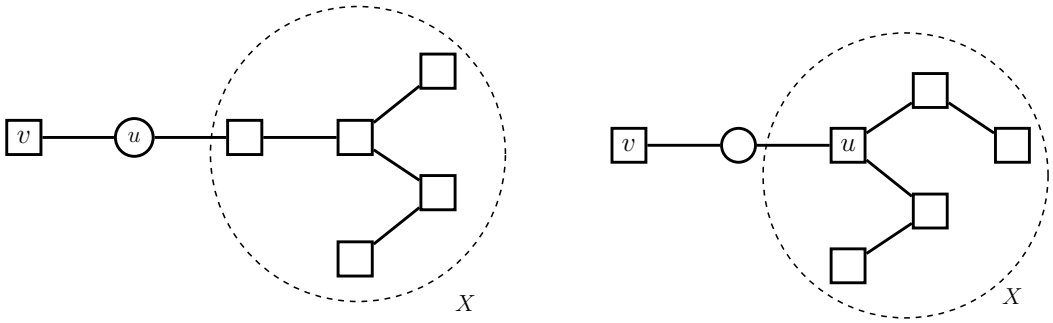


Figure 5.2: An illustration of the optimal-decomposition property for recursion (5.2), terminal vertices are denoted by rectangles and non-terminal vertices are denoted by circles. In Case (a), a leaf vertex v is connected to X through a vertex $u \in V \setminus X$ such that degree of u is $\mathsf{deg}_T(u) \geq 2$. In Case (b), a leaf vertex v is connected to X through a vertex $u \in X$ such that degree of vertex u is $\mathsf{deg}_T(u) \geq 2$.

With Lemma 5.1 in place the length of a Steiner tree for terminal set K can be computed as $f_v(K \setminus \{v\})$ for some $v \in K$.

Theorem 5.2 (Dreyfus–Wagner [34])**.** *The length of a Steiner tree can be computed in* $O(3^k n + 2^k n^2 + n^2 \log n + nm)$ *steps.*

Proof. The correctness of the algorithm immediately follows the proof of Lemma 5.1. To verify the complexity of Algorithm 2, computing shortest path can be done in $O(n \log n + m)$ using Dijkstra's algorithm for n times which makes the complexity of initialisation $O(n^2 \log n + nm)$ (see Lemma 4.6).

The complexity of recursion (5.1) in Lemma 5.1 is bounded by number of possibilities of choosing v, X and X'. Every terminal $t \in K$ belongs to exactly one of the subsets X', $X \setminus X'$ or $K \setminus X$. Hence the number of possibilities is bounded by $O(3^k n)$. The complexity of recursion (5.2) in Lemma 5.1 is bounded by number of possibilities of choosing v and X. Hence the number of tuples for which recursion (5.2) is carried out is bounded by $O(2^k n)$ making the overall complexity of the algorithm $O(3^k n + 2^k n^2 + n^2 \log n + nm)$. □

The Dreyfus–Wagner algorithm only returns the weight of a Steiner tree. However, the algorithm can be modified to construct a Steiner tree by keeping track of vertex-subset pair which satisfies the optimal-decomposition property and adding a Traceback procedure in the end. We will discuss this in the next section.

Algorithm 2: DREYFUS–WAGNER

Input: A network $N = (V, E, w)$ and a terminal set K.
Output: The length of a Steiner tree.

```
// Initialization
for v ∈ V do
    (d, p) ← Dijkstra(N, v)
end
for v ∈ V do
    for u ∈ K do
        f_v({u}) ← d(v, u)
        b_v({u}) ← {(u, {u})}
    end
end
for m = 2 to |K| − 1 do
    for X ⊆ K with |X| = m do
        // Recursion (5.1) from Lemma 5.1
        for v ∈ V do
            for X′ ⊂ X and X′ ≠ ∅ do
                if f_v(X′) + f_v(X \ X′) < g_v(X) then
                    g_v(X) ← f_v(X′) + f_v(X \ X′)
                    b_v(X) ← {(v, X′), (v, X \ X′)}
                end
            end
        end
        // Recursion (5.2) from Lemma 5.1
        for v ∈ V do
            for u ∈ X do
                if d(v, u) + f_u(X \ {u}) < f_v(X) then
                    f_v(X) ← d(v, u) + f_u(X \ {u})
                    b_v(X) ← {(u, X \ {u})}
                end
            end
            for u ∈ V \ X do
                if d(v, u) + g_u(X) < f_v(X) then
                    f_v(X) ← d(v, u) + g_u(X)
                    b_v(X) ← {(u, X)}
                end
            end
        end
    end
end
return f_v(K \ {v}), for some v ∈ K
```

5.4 Extracting a Steiner tree

In this section, we discuss a procedure to extract a Steiner tree using the additional bookkeeping information stored while solving the recursions (5.1) and (5.2). Let $b_v(X)$ denote a set of vertex-subset pair satisfying the optimal-decomposition property for a Steiner tree connecting vertices in $X \cup \{v\}$ (see Lines 16, 24, 30 in Algorithm 2). The Steiner tree T for the terminal set K is automatically constructed in the course of solving recursions (5.1) and (5.2) iteratively by standard method without any extra computation provided that care is taken not to change $b_v(X)$ from one iteration to the next unless this strictly reduces the appropriate cost. The pseudo-code of a procedure to construct a Steiner tree from the bookkeeping information is available in Procedure 6. Using this procedure we can construct a Steiner tree as $\textsc{Traceback}(v, K \setminus \{v\})$ for some $v \in K$.

Procedure 6: $\textsc{Traceback}(v, X)$

```
1 if b_v(X) = {(u, X)} then
2 |   return {v, u} ∪ Traceback(u, X)
3 else
4 |   return ⋃_{(u,X')∈b_v(X)} Traceback(u, X')
5 end
```

5.5 The edge-linear algorithm

As a type of exact algorithm Erickson, Monma and Veinott [38] presented a parameterised algorithm which runs in edge-linear time for solving the Steiner problem in graphs. An edge-linear time algorithm for the Steiner problem has an advantage of running in linear time with respect to the size of input graph even though the problem is $\mathcal{NP}$-complete and the exponential factor of the algorithm is restricted to number of terminals. An algorithm of similar nature was presented by Hougardy, Silvanus and Vugen [63] in 2014. Both algorithms use dynamic programming and find a Steiner tree in $O(3^k n + 2^k(n \log n + m))$ time-steps.

The pseudo-code of the Erickson–Monma–Veinott algorithm is listed in Algorithm 3. In this thesis, the term *edge-linear algorithm* will only be used to refer the Erickson–Monma–Veinott algorithm for solving the Steiner problem in graphs.

Erickson, Monma and Veinott [38] presented simplifications on top of the Dreyfus–Wagner algorithm to reduce the number of computations. There are three simplifications of the calculation associated with each pair (v, X). Firstly, it is sufficient to split set X only when vertex $v \in V \setminus X$. Secondly, while splitting set X into subsets X' and $X \setminus X'$ if there is no terminal vertex in $X \setminus X'$ then without loss of generality we can restrict the computation only to X' that contain any given node in X and finally, compute the shortest path distance between vertices only on demand when required. In this way there is no need to find the minimum distance of all pair of vertices in the graph. Using these modifications, the Dreyfus–Wagner recursion reduces to,

$$g_v(X) = \min_{\emptyset \neq X' \subset X} \{f_v(X') + f_v(X \setminus X')\} \tag{5.3}$$

$$f_v(X) = \min_{u \in V \setminus X} \{d(v, u) + g_u(X)\}. \tag{5.4}$$

The key idea of the algorithm is to batch the computation of $f_v(X)$ for all vertices $v \in V \setminus X$ for a fixed X while computing the single source shortest path. This can be done by constructing a network $N' = (V', E', w')$ such that,

$$V' = V \cup \{s\}$$

$$E' = E \cup \bigcup_{v \in V} \{s, v\}$$

$$w'(e) = \begin{cases} w(e), & \text{if } e \in E, \\ f_u(X \setminus \{u\}), & \text{if } e = \{s, u\} \text{ and } u \in X, \\ g_u(X), & \text{if } e = \{s, u\} \text{ and } u \in V \setminus X. \end{cases}$$

The aforementioned network construction is illustrated in Figure 5.3. Now, if we invoke Dijkstra's algorithm on network N' with s as the source vertex then the cost of a shortest path from s to $v \in V$ is exactly,

$$f_v(X) = d_{N'}(s, v) = \min\{\min_{u \in X}\{d(u, v) + f_u(X \setminus \{u\})\}, \min_{u \in V \setminus X}\{d(v, u) + g_u(X)\}\}.$$

Hence, one invocation of Dijkstra's algorithm evaluates the cost $f_v(X)$ of a Steiner tree connecting vertices $X \cup \{v\}$ for all $v \in V$.

The complexity of recursion (5.3) is bounded by number of possibilities of choosing v, X and X'. Every terminal $t \in K$ belongs to exactly one of the subsets X', $X \setminus X'$ or $K \setminus X$. Hence, the number of possibilities is

bounded by $O(3^k n)$. The complexity of recursion in (5.4) is bounded by the number of possibilities of choosing X. Hence, the number of tuples for which recursion (5.4) is carried out is bounded by $O(2^k)$ and the distances between vertices are computed on demand. From Lemma 4.6, a shortest path can be computed in $O(n \log n + m)$ time-steps which makes the overall complexity of the algorithm $O(3^k n + 2^k(n \log n + m)$ time-steps.

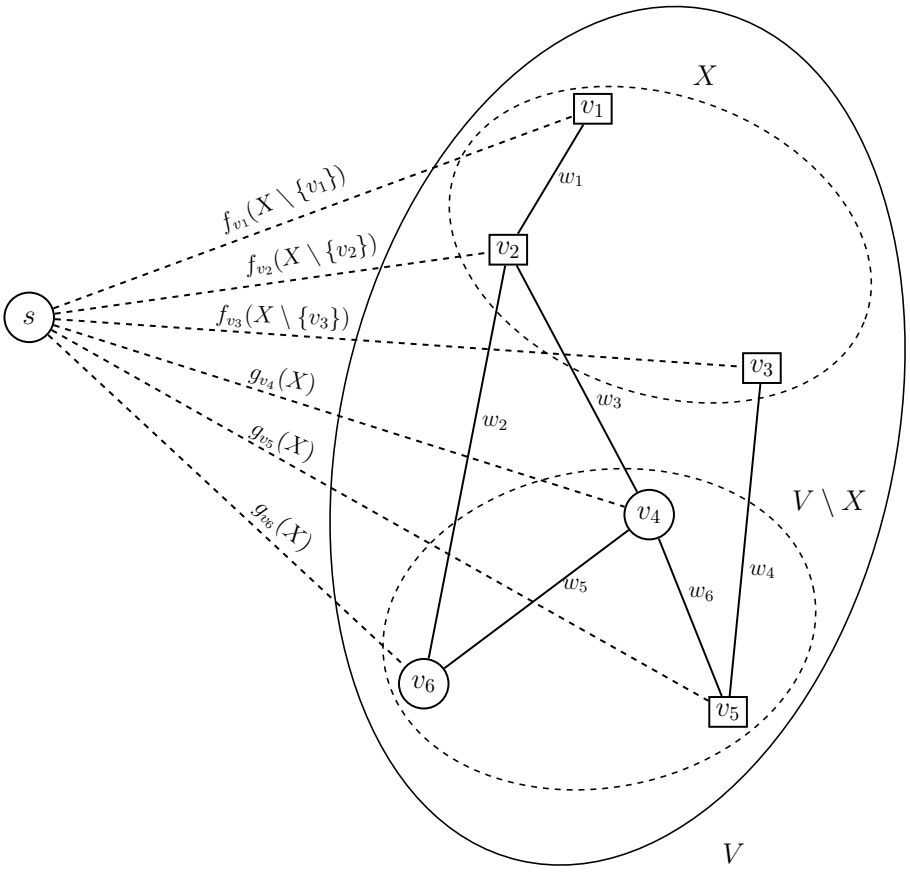


Figure 5.3: An illustration of network $N' = (V', E', w')$ construction using $N = (V, E, w)$ with terminal set $K = \{v_1, v_2, v_3, v_5\}$ and subset $X = \{v_1, v_2, v_3\}$ for batching the computations of $f_v(X)$ for all $v \in V$ in the Erickson–Monma-Veinott algorithm for solving the Steiner problem. The terminal vertices are denoted by rectangles and non-terminal vertices are denoted by circles. The vertex set V is partitioned into two subsets X and $V \setminus X$ and it is separated using two dotted regions. We add an additional vertex $s \notin V$ and artificial edges $\{s, u\}$ for all $u \in V$ with an edge weight $g_u(X)$ if $u \in V \setminus X$, $f_u(X \setminus \{u\})$ otherwise. The artificial edges are denoted by dotted lines.

Algorithm 3: Erickson–Monma–Veinott

Input: A network $N = (V, E, w)$ and a terminal set K.
Output: The length of a Steiner tree.

```
1  for u ∈ K do
2  |  (d, p) ← Dijkstra(N, u)
3  |  for v ∈ V do
4  |  |  f_v({u}) ← d(v), b_v({u}) ← {(u, {u})}
5  |  end
6  end
7  for m = 2 to |K| − 1 do
8  |  for X ⊆ K with |X| = m do
   |  |  // Recursion (5.3)
9  |  |  for v ∈ V do
10 |  |  |  for X′ ⊂ X and X′ ≠ ∅ do
11 |  |  |  |  if f_v(X′) + f_v(X \ X′) < g_v(X) then
12 |  |  |  |  |  g_v(X) ← f_v(X′) + f_v(X \ X′)
13 |  |  |  |  |  b_v(X) ← {(v, X′), (v, X \ X′)}
14 |  |  |  |  end
15 |  |  |  end
16 |  |  end
   |  |  // Recursion (5.4)
17 |  |  V′ ← {s} ∪ V, E′ ← ⋃_{v∈V} {s, v} ∪ E
18 |  |  for e ∈ E′ do
19 |  |  |  if e ∈ E then
20 |  |  |  |  w′(e) ← w(e)
21 |  |  |  else if e = {s, u} and u ∈ X then
22 |  |  |  |  w′(e) ← f_u(X \ {u})
23 |  |  |  else if e = {s, u} and u ∈ V \ X then
24 |  |  |  |  w′(e) ← g_u(X)
25 |  |  |  end
26 |  |  end
27 |  |  N′ ← (V′, E′, w′)
28 |  |  (d, p) ← Dijkstra(N′, s)
29 |  |  for v ∈ V \ X do
30 |  |  |  f_v(X) ← d(v),  u ← p(v)
31 |  |  |  if u ≠ s then
32 |  |  |  |  b_v(X) ← {(u, X)}
33 |  |  |  end
34 |  |  end
35 |  end
36 end
37 return f_v(K \ {v}) for some v ∈ K
```

5.6 Solving the group Steiner problem as the Steiner problem

A transformation for solving the group Steiner problem as the Steiner problem was introduced by Voß [122] in 1990 and it is also mentioned in a book by Hwang, Richards and Winter [66]. Even though the transformation is simple it was ignored for a long time. The transformation was restated and a practical implementation was presented by Duin, Volgenant and Voß [37] in 2004. Using this transformation most algorithms used to solve the Steiner problem can also be used to solve the group Steiner problem in graphs. However, in this thesis we mainly focus on scalable algorithms for the group Steiner problem, as a result, we use the Erickson–Monma–Veinott discussed in Section 5.5 for solving the Steiner problem and thus for the group Steiner problem too. The transformation presented in this section is the work of Voß [122]. Additionally, we present the proof of the reduction in Lemma 5.4.

Let us recall the Group Steiner Problem (GSP), given a connected network $N = (V, E, w)$ and a collection $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$ of subsets of V called groups. The task is to find a minimum-weight tree connecting at least one vertex from each group $Q_i \in \mathcal{Q}$.

Let $C = W(N) = \sum_{e \in E} w(e)$ denote the weight of the network N. The GSP instance can be transformed into SP instance in linear-time by constructing a network $N' = (V', E', w')$ and terminal set K such that,

$$K = \{q_1, \ldots, q_k\}$$

$$V' = V \cup K$$

$$E' = E \cup \{\{q_i, v\} | v \in Q_i \text{ for all } Q_i \in \mathcal{Q}\}$$

$$w'(e) = \begin{cases} w(e), & \text{if } e \in E, \\ 2C, & \text{if } e \in E' \setminus E, \\ \infty, & \text{otherwise.} \end{cases}$$

The transformation from the group Steiner problem to the Steiner problem is illustrated in Figure 5.4.

Using this transformation, we can employ any known algorithm to solve the Steiner problem in N'. We shall prove that by finding a Steiner tree T' in N' we obtain a group Steiner tree T in N by removing the terminals K and edges connecting K in T'.

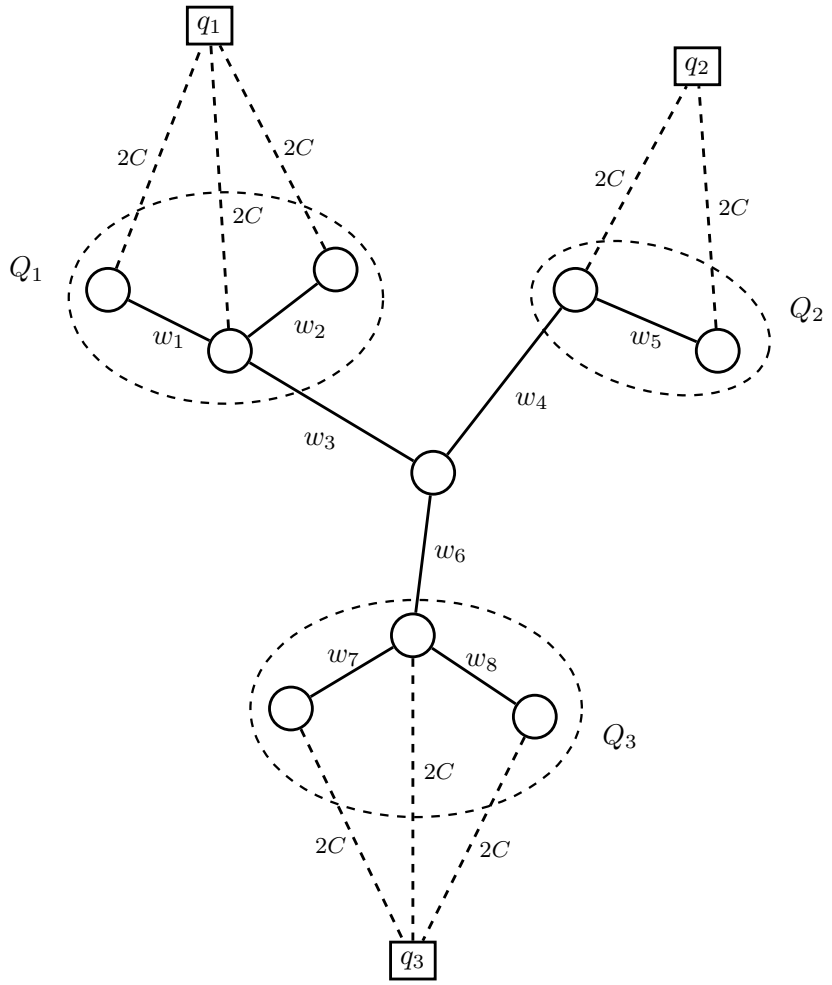


Figure 5.4: An illustration of graph construction for solving the group Steiner problem as the Steiner problem. The terminal vertices are denoted by rectangles and the non-terminal vertices are denoted by circles. The vertices inside a dotted region form a group. In this example, we have three groups $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$. A terminal vertex q_i is added for each group $Q_i \in \mathcal{Q}$. An artificial edge is connected from each vertex in a group $v \in Q_i$ to the terminal vertex q_i with edge weight $2C$ where $C = W(N)$ and the artificial edges are denoted by dotted lines.

Lemma 5.3. A Steiner tree T' in network N' with k terminals has exactly k leaf vertices and all leaf vertices in T' are terminals.

Proof. Let T' be a Steiner tree in N' such that T' connects all the terminals in K and $k = |K|$. Let u be a leaf node in T' such that $u \notin K$ then by removing

u and the unique edge incident to u, we get T'' such that $W(T'') < W(T')$. However, this contradicts our hypothesis that T' is minimum. Hence, all the leaf vertices are terminals.

Let u be a terminal vertex in T' such that $\mathsf{deg}_T(u) \geq 2$ and $v, w \notin K$ be any two vertices adjacent to u. Since N is connected, from Proposition 2.2 there exists a path P in N with length $W(P) \leq C$ where $C = W(N)$, which is clearly less than $2C$ (see Proposition 2.2). So the edge between vertices u and w can be replaced by path P in T' to get T'' such that $W(T'') < W(T')$. However, this contradicts the minimality of T'. Hence, there exists no terminal vertex with degree greater than one in T'. Furthermore, a Steiner tree in N' has exactly k leaf vertices and all the leaf vertices are terminals. □

Lemma 5.4. Let T' be a group Steiner tree in network N' with terminals K then the graph obtained by removing terminals K and edges connecting the terminals in T' is a group Steiner tree in network N.

Proof. Let T' be a Steiner tree in N' connecting all the terminals in K and $k = |K|$ then we prove that the group Steiner tree T for N can be obtained from T' by removing terminal vertices $q_i \in K$ and edges connecting q_i for all $i = 1, \ldots, k$. The transformation from T' to T can be done in linear time.

From Lemma 5.3 every terminal vertex $q_i \in K$ is a leaf vertex in T'. Furthermore, from the construction of N' each terminal vertex q_i is connected only through vertices $v \in Q_i$. So, there exists at least one vertex from each group Q_i for all $i = 1, \ldots, k$ in $V(T')$.

Let us assume that T' has exactly x vertices such that $x \geq k$ then from Proposition 2.5, T' has exactly $x - 1$ edges. From Lemma 5.3 every terminal vertex $q_i \in K$ is a leaf. The resultant graph obtained by removing a leaf vertex q_i and the unique edge incident to q_i remains connected. Clearly, in the transformation from T' to T we remove exactly k leaf vertices and k edges. Hence T is a connected graph with $x - k$ vertices and $x - k - 1$ edges. From Proposition 2.6 T is a tree.

For the sake of contradiction, let us assume that there exists a group Steiner tree T_s in N such that $W(T_s) < W(T)$, then T can be replaced by T_s in T' to get T'_s such that $W(T'_s) < W(T')$. However, this contradicts the minimality of T'. Hence, T is the group Steiner tree in N connecting at least one vertex from each group $Q_i \in \mathcal{Q}$. □

Chapter 6

Parallel implementation of the edge-linear algorithm

In the previous chapter, we discussed the Erickson–Monma–Veinott algorithm for solving the Steiner problem in graphs. In practice, the algorithm should be implemented in software to verify its practical scalability on modern-computer architectures. To examine the performance of an algorithm implementation, it is often sufficient to measure the runtime, memory bandwidth and memory usage. However, it is also important to take into consideration the architecture details and the peak-achievable bandwidth of the hardware. We verify the scalability of our implementation of the Erickson–Monma–Veinott algorithm in terms of its runtime, memory bandwidth and peak-memory usage. Recall that, we used the term the edge-linear algorithm to refer the Erickson–Monma–Veinott algorithm. In what follows, the use of term *runtime* refers to actual wall-clock time of the experiments.

Knowing that the Steiner problem is $\mathcal{NP}$-complete and unavailability of an efficient exact-algorithm implementation most solutions for the Steiner problem adopt to approximation algorithms. For example, Bhalotia *et al.* [9]; Ding *et al.* [32]; He *et al.* [60]; Kacholia *et al.* [75]; Lappas, Liu and Terzi [88]; Rozenshtein *et al.* [106] have used Steiner tree approximation algorithms for solving the data-mining problems. In this chapter, we present an implementation of the edge-linear algorithm for the Steiner problem which can scale easily provided that the number of terminals is small. We also explore possible design choices to parallelise the edge-linear algorithm to achieve speed-up in computations. The efficiency of our implementation of the edge-linear algorithm relies on the efficiency of Dijkstra's algorithm because it uses Dijkstra's algorithm as a subroutine to compute shortest paths (see Algorithm 3). For

this reason, we present a scalable implementation of Dijkstra's algorithm. Additionally, we present an implementation of the binary and Fibonacci heaps.

Our software is written in C programming language (C99) using OpenMP API [100] (`-fopenmp`) for parallelisation. We compile our source code using `gcc` compiler with optimisation level `-O5` and `-march=native` options to enable architecture-specific optimisations. In this work, we focus on engineering the Erickson–Monma–Veinott algorithm for Intel micro-processors with 64-bit Haswell microarchitecture. The term *word* refers to a sequence of 64-bits and the term *cache line* refers to eight-consecutive words or a group of 512-bits. The use of term *throughput* refers to the memory bandwidth which is the data-transfer rate between the main memory and the processor. The data-type `index_t` is a 64-bit signed integer. The experimental version of our current implementation is available as open source [115].

6.1 Implementation challenges

We address the following engineering challenges to design a scalable implementation of the edge-linear algorithm for solving the Steiner problem, and by reduction, the group Steiner problem:

1. *Memory consumption.* The edge-linear algorithm has exponential space complexity with respect to the number of terminals k and the working memory should be in the order of $O(2^k n+n+m)$, where n is the number of vertices and m is the number of edges in the graph.

2. *Memory interface.* In Intel documentation [69, 70] it is reported that, a Haswell processor fetches a complete cache line (8 words) from the main memory every time a new memory-read request has been made. Hence, the memory accesses are efficient only if as many as possible words in the cache line are utilised for the computations.

3. *Graph traversal.* Each instance of the Dijkstra procedure traverses all the edges in a graph input which essentially results in an arbitrary pattern of memory accesses to the main memmory (see Section 4.8). As a consequence the data access patterns tend to have less cache locality if the input graphs are large.

4. *Parallel execution.* A single execution core is generally not sufficient to saturate the complete memory bandwidth, especially if multiple channels to the main memory are available (see Section 7.3).

We address these implementation challenges as follows. First, let us recall the Erickson–Monma–Veinott algorithm for the Steiner problem. The algorithm makes use of the optimal-decomposition property to compute a Steiner tree. The core of such a design is the 2^k executions of Dijkstra's algorithm for computing shortest paths (see Section 5.5). To reduce the memory consumption, we use natural-bit representation of subsets which is a classical technique in algorithm engineering (Knuth [82, Section 3B]). In the context of graph representation and memory interface, it is possible to take advantage of optimisation techniques such as: memory prefetching and optimising the cache line level by utilising as many as possible words in the cache line. Modern microprocessors perform memory prefetching to hide the latency of memory accesses, in this case a single cache miss would bring in multiple cachelines that would subsequently be accessed which result in a cache hit (Intel [69, 70]; Lumsdaine *et al.* [92]). To make effective use of these optimisation techniques, we follow the array of arrays representation for graphs from Mehta and Shani [93, Section 2.2]. In this representation, the adjacency data of each vertex is stored in an array and therefore in contiguous memory locations, which reduces the cache pollution and increases the cache locality (Park, Penner and Prasanna [102]).

In modern microprocessors the arithmetic instructions are pipelined, meaning multiple instructions of same type can be executed simultaneously by being in different stages of the pipeline provided that there is no data dependency. If not enough independent instructions are available for execution then the pipeline stalls. Ignoring the latency of memory accesses, to saturate the arithmetic bandwidth a software design should ensure that each core can execute a sufficient number of independent instructions.

Intel Haswell CPUs have substantial memory access latency ranging from less than ten clock cycles for L1 cache hit to hundreds of clock cycles for main memory accesses (Intel [89]). An important aspect for achieving high performance is to reduce the number of main-memory accesses. In the case of our implementation of the Erickson–Monma–Veinott algorithm, avoiding main-memory accesses is not possible because of the large problem instances. To hide the latency of memory accesses and to keep the pipeline busy each core should execute a sufficient number of memory-read and memory-write requests. In our current implementation, to better saturate the memory bandwidth and to benefit from multiple-execution cores we parallelise the computations over the subsets of the terminal set K.

6.2 Implementation of priority queues

In this section, we present an implementation of binary and Fibonacci heaps. Our implementation of binary heap is fairly simple. We implement a n-element binary heap using a n-element one-dimensional array; a binary heap element at index $1 \leq i \leq n$ has its left child at index $2i$ and right child at index $2i + 1$, consequently the parent of an element at index $i \geq 2$ is located at index $\lfloor i/2 \rfloor$.

On the other hand, implementing a Fibonacci heap is challenging when one tries to achieve high memory bandwidth performance. In our implementation, a Fibonacci-heap element is represented using an explicit pointer structure and the definition is available in Listing 6.1. Each element of the Fibonacci heap contains four pointers: a pointer to its parent element, a pointer to its left sibling, a pointer to its right sibling and a pointer to an arbitrary child. In addition, it contains a key, vertex number (value), rank of the element and a field indicating whether the element is marked.

```
1  typedef struct fheap_node {
2      struct fheap_node *parent;
3      struct fheap_node *left;
4      struct fheap_node *right;
5      struct fheap_node *child;
6      index_t rank;
7      index_t marked;
8      index_t key;
9      index_t vertex_no;
10 } fheap_node_t;
```

Listing 6.1: A structure definition of a Fibonacci heap element. Each element has four pointers: a `parent` pointer which points to its parent element, a `left` pointer which points to its left sibling, a `right` pointer which points to its right sibling and a `child` pointer which points to an arbitrary child. In addition, the `rank` field stores the rank of the element, the `key` field stores the key (priority),and the `vertex_no` field stores the value of element. The `marked` field is set if the element is marked, more precisely to check if there was a previous CUT operation performed on the element.

In our implementation, the Fibonacci heap data-structure contains an array of pointers to all root-nodes and an array of pointers to all the nodes in the heap. In addition, it stores the maximum (upper limit) number of trees, the maximum (upper limit) number of elements and the current number of elements in the heap. The definition of the Fibonacci heap data-structure is available in Listing 6.2.

```
1  typedef struct fheap {
2      fheap_node_t **trees;
3      fheap_node_t **nodes;
4      index_t max_nodes;
5      index_t max_trees;
6      index_t n;
7  } fheap_t;
```

Listing 6.2: A structure definition of a Fibonacci heap data-structure. The structure contains an array of pointers named `trees`, which point to all root elements in the heap and an array of pointers named `nodes`, which point to all elements in the heap. In addition, it contains a field `max_nodes` to store the maximum (upper limit) number of elements in heap, a field `max_trees` to store the maximum (upper limit) number of trees and a field `n` to store the current number of elements in heap.

The efficiency of Dijkstra's algorithm relies on the asymptotic complexity of the priority-queue operations (see Section 4.8). The theoretical difference between the binary and Fibonacci heap is that, Fibonacci heap supports Decrease-key and Insert priority-queue operations in amortised constant time compared to $O(\log n)$ time in binary heap. The improved asymptotic complexity of the Decrease-key operation is important for the theoretical performance. However, from an implementation perspective the decrease in time complexity of the Insert operation is mostly inconsequential as it does not affect the runtime of Dijkstra's algorithm significantly, unless n is too large ($n > m$). On the other hand, we can potentially benefit from the improved time complexity of the Decrease-key operation (Cormen *et al.* [24, Chapter 19]). Hence, an optimal implementation of the Decrease-key operation is of at most importance. We employ the algorithms discussed in Section 4.4 to implement the priority-queue operations of the Fibonacci heap. A implementation of Decrease-key and Merge priority-queue operations is available in Listing 6.3 and Listing 6.4, respectively.

6.3 Implementation of Dijkstra's algorithm

In this section, we present an implementation of visit-and-label algorithm designed by Dijkstra for solving the shortest path problem in graphs. Using Fibonacci heap the asymptotic complexity of Dijkstra's algorithm reduces to $O(m + n \log n)$ time-steps. The proof of Dijkstra's algorithm is presented in Lemma 4.6 and the pseudo-code is available in Algorithm 1. An implementation of Dijkstra's algorithm is available in Listing 6.5.

```
1  void decrease_key(fheap_t *heap, index_t vertex_no, index_t new_key)
2  {
3      fheap_node_t *cut, *parent_node, *new_roots;
4      fheap_node_t *right_node, *left_node;
5      index_t prev_rank;
6      cut         = heap->nodes[vertex_no];
7      parent_node = cut->parent;
8      cut->key    = new_key;
9
10     if(!parent_node) return;
11
12     left_node        = cut->left;
13     right_node       = cut->right;
14     left_node->right = right_node;
15     right_node->left = left_node;
16     cut->left        = cut;
17     cut->right       = cut;
18     new_roots        = cut;
19     while(parent_node && parent_node->marked) {
20         parent_node->rank--;
21         if(parent_node->rank) {
22             if(parent_node->child == cut)
23                 parent_node->child = right_node;
24         }
25         else {
26             parent_node->child = NULL;
27         }
28         cut = parent_node;
29         parent_node      = cut->parent;
30         left_node        = cut->left;
31         right_node       = cut->right;
32         left_node->right = right_node;
33         right_node->left = left_node;
34         left_node        = new_roots->left;
35         new_roots->left  = cut;
36         left_node->right = cut;
37         cut->left        = left_node;
38         cut->right       = new_roots;
39         new_roots        = cut;
40     }
41     if(!parent_node) {
42         prev_rank = cut->rank + 1;
43         heap->trees[prev_rank] = NULL;
44         heap->value -= (1 << prev_rank);
45     }
46     else {
47         parent_node->rank--;
48         if(parent_node->rank) {
49             if(parent_node->child == cut)
50                 parent_node->child = right_node;
51         }
52         else {
53             parent_node->child = NULL;
54         }
55         parent_node->marked = 1;
56     }
57     heap_merge(heap, new_roots);
58 }
```

Listing 6.3: An implementation of DECREASE-KEY operation in Fibonacci heap.

```
1  void heap_merge(fheap_t *heap, fheap_node_t *tree_list)
2  {
3      fheap_node_t *first_node, *next_node, *left_child, *right_child;
4      fheap_node_t *ptr, *new_root, *temp, *temp2,
5      index_t rank;
6      ptr = first_node = tree_list;
7      do {
8          next_node   = ptr->right;
9          ptr->left   = ptr;
10         ptr->right  = ptr;
11         ptr->parent = NULL;
12         new_root    = ptr;
13         rank = ptr->rank;
14         do {
15             if((temp = heap->trees[rank])) {
16                 heap->trees[rank] = NULL;
17                 heap->value      -= (1 << rank);
18                 if(temp->key < new_root->key) {
19                     temp2    = new_root;
20                     new_root = temp;
21                     temp     = temp2;
22                 }
23                 if(rank++ > 0) {
24                     right_child  = new_root->child;
25                     left_child   = right_child->left;
26                     temp->left   = left_child;
27                     temp->right  = right_child;
28                     left_child->right = temp;
29                     right_child->left = temp;
30                 }
31                 new_root->child = temp;
32                 new_root->rank  = rank;
33                 temp->parent    = new_root;
34                 temp->marked    = 0;
35             }
36             else {
37                 heap->trees[rank]   = new_root;
38                 heap->value        += (1 << rank);
39                 new_root->marked = 1;
40             }
41         } while(temp);
42         ptr = next_node;
43     } while(ptr != first_node);
44 }
```

Listing 6.4: An implementation of MERGE priority queue operation in Fibonacci heap.

Given a network $N = (V, E, w)$ and a source vertex $s \in V$. For all $v \in V$, let $d(v)$ denote the length of a shortest path from s to v computed by the algorithm. Dijkstra's algorithm essentially partitions the vertex set V into queued Q and visited $V \setminus Q$ partitions. In each iteration of the while loop (Lines 15–29 in Listing 6.5), a queued vertex $u \in Q$ with minimum distance $d(u)$ is selected and marked as visited. Furthermore, all the edges incident to u are relaxed by setting the distance $d(v) = \min\{d(v), d(u) + w(\{u, v)\}$ for each vertex v adjacent to u such that $v \in Q$(see Section 4.8).

Our implementation accepts the input graph in adjacency-list format and computes the distance of shortest paths from a single-source vertex to all other vertices int he graph. Our implementation of Dijkstra's algorithm is sequential, meaning it runs on a single execution core. However, we shall see that multiple independent instances of the Dijkstra procedure can be executed on a multi-core CPU to saturate the memory interface.

```
1  void dijkstra(index_t s, index_t n, index_t *pos,
2                index_t *adj, index_t *d, index_t *visit)
3  {
4      heap_t *heap = heap_alloc(n);
5      // initialisation
6      for(index_t v = 0; v < n; v++) {
7          d[v]     = MAX_DISTANCE;
8          visit[v] = 0;
9      }
10
11     d[s] = 0;
12     for(index_t v = 0; v < n; v++)
13         heap_insert(heap, v, d[v]);
14     // visit and label
15     while(heap->n > 0) {
16         index_t u = heap_delete_min(heap);
17         visit[u]  = 1;
18         index_t pos_u  = pos[u];
19         index_t *adj_u = adj + pos_u;
20         index_t n_u    = adj_u[0];
21         for(index_t i = 1; i <= 2*n_u; i += 2) {
22             index_t v   = adj_u[i];
23             index_t d_v = d[u] + adj_u[i+1];
24             if(!visit[v] && d[v] > d_v) {
25                 d[v] = d_v;
26                 heap_decrease_key(heap, v, d_v);
27             }
28         }
29     }
30     heap_free(heap);
31 }
```

Listing 6.5: An implementation of Dijkstra's algorithm.

6.4 Implementation of the edge-linear algorithm

In this section, we present our implementation of the Erickson–Monma–Veinott algorithm for solving the Steiner problem in graphs. Recall that, we referred the Erickson–Monma–Veinott algorithm as the edge-linear algorithm. The algorithm is described in Section 5.5 and its pseudo-code is

available in Algorithm 3. The source-code of our implementation is available in Listing 6.6.

Subset generation. We use classical natural-bit representation [82, Section 3B] to denote a subset X of the terminal set $K = \{q_1, q_2, \ldots, q_k\}$. More precisely, we use k bits to represent X in which we set $i-1$th bit to 1 if the terminal vertex $q_i \in X$, 0 otherwise. This defines a bijection $h : 2^K \to \{0, 1, \ldots, 2^k - 1\}$ where 2^K is the powerset of K. Indeed, X is represented with an integer value in range $0, \ldots, 2^k - 1$. The implementation of the edge-linear algorithm requires generating all p-subsets of K for each $1 < p \leq k$. We use lexicographic bit-permutation generator using bit-twiddling hacks to generate these subsets from Anderson [3] and Warren [126]. Our implementation uses 64-bit indexing and therefore the maximum supported terminal-set size is 64. However, we should keep in mind that the algorithm requires exponential memory resources with respect to k and its space complexity is $O(2^k n + n + m)$.

Memory layout. Let us recall from Section 5.3 that given a subset X of the terminal set K and a vertex v. Let $g_v(X)$ denotes the cost of a Steiner tree connecting the vertices in $X \cup \{v\}$ such that degree of vertex v is at least 2. Let $f_v(X)$ denote the cost of a Steiner tree connecting the vertices in $X \cup \{v\}$. To store the values of $g_v(X)$ and $f_v(X)$ we use a one-dimensional array of size $2^k \cdot n$, where n is the number of vertices and k is number of terminals in the input graph. More precisely, for each subset $X \subseteq K$ and vertex v, we store the cost of a Steiner tree connecting $X \cup \{v\}$ at index $h(X) \cdot n + v$ with subset major index, where $h : 2^K \to \{0, 1, \ldots, 2^k - 1\}$. We organise the computations efficiently by computing $g_v(X)$ prior to computing $f_v(X)$ using the same memory space. Hence, there is no additional memory required for computing $g_v(X)$. For tracking the Steiner tree we use a one-dimensional array of size $2^{k+1} \cdot n$. For each subset $X \subseteq K$ and vertex v, we store the vertex u and subset $X' \subseteq X$ satisfying the optimal-decomposition property at index $h(X) \cdot 2n + 2v$ and $h(X) \cdot 2n + 2v + 1$, respectively (see Section 5.4).

Parallelisation over subsets. Our implementation of the edge-linear algorithm for solving the Steiner problem relies on parallelisation of a single block over the subsets of the terminal set K using OpenMP API via `omp parallel for` construct with default scheduling. The block parallelise the computations of $g_v(X)$ and $f_v(X)$ over all p-subsets $X \subseteq K$ for each $1 < p \leq k$. We compute the values of $g_v(X)$ and $f_v(X)$ from bottom-up starting from subsets with size $|X| = 2$ to $|X| = k$ by iteratively increasing the subset size. Furthermore, the values of $g_v(X)$ and $f_v(X)$ with subset size $p = |X|$ depend only on $f_v(X')$ such that $\emptyset \neq X' \subset X$, indeed $|X'| < p$. As a result, we can

```
1  void emv_kernel(index_t n, index_t k, index_t *kk, index_t *f_v, index_t *pos,
2                  index_t *adj, index_t *d, index_t *visit, index_t nt)
3  {
4      for(index_t q = 0; q < k; q++) {
5          dijkstra(kk[q], n+1, pos, adj, d, visit);
6          index_t *f_q = f_v + FV_INDEX(0, n, k, 1<<q);
7  #pragma omp parallel for
8          for(index_t v = 0; v < n; v++)
9              f_q[v] = d_th[v];
10     }
11
12     for(index_t p = 2; p < k; p++) {
13         index_t kCp  = choose(k, p);
14         index_t *X_a = (index_t *) MALLOC(kCm * sizeof(index_t));
15         index_t i = 0; index_t z = 0;
16         // bit twiddling hacks: generating bit-permutations
17         for(index_t X = (1<<p)-1; X < (1<<k);
18             z = X|(X-1), X = (z+1)|(((~z & -~z)-1) >> (__builtin_ctz(X) + 1))) {
19             X_a[i++] = X;
20         }
21         index_t block_size = kCp/nt;
22 #pragma omp parallel for
23         for(index_t th = 0; th < nt; th++) {
24             index_t start     = th*block_size;
25             index_t stop      = (th == nt-1) ? kCp-1 : (start+block_size-1);
26             index_t *d_th     = d + (n+nt)*th;
27             index_t *visit_th = visit + (n+nt)*th;
28             for(index_t i = start; i <= stop; i++) {
29                 index_t X    = X_a[i];
30                 index_t *f_X = f_v + FV_INDEX(0, n, k, X);
31                 index_t Xd   = 0;
32                 // bit twiddling hacks: generating proper subsets of X
33                 for(Xd = X & (Xd - X); Xd != X; Xd = X & (Xd - X)) {
34                     index_t X_Xd    = (X & ~Xd); // X - X'
35                     index_t *f_Xd   = f_v + FV_INDEX(0, n, k, Xd);
36                     index_t *f_X_Xd = f_v + FV_INDEX(0, n, k, X_Xd);
37                     for(index_t v = 0; v < n; v++)
38                         f_X[v] = MIN(f_X[v], f_Xd[v] + f_X_Xd[v])
39                 }
40                 // graph reconstruction
41                 index_t s = n + th; index_t ps = pos[s];
42                 index_t *adj_s = adj + (ps+1);
43                 for(index_t u = 0; u < n; u++)
44                     adj_s[2*u+1] = f_X[u];‘
45                 for(index_t q = 0; q < k; q++) {
46                     if(!(X & (1<<q))) continue;
47                     index_t u = kk[q]; index_t X_u = (X & ~(1<<q));
48                     index_t i_X_u = FV_INDEX(u, n, k, X_u);
49                     adj_s[2*u+1]  = f_v[i_X_u];
50                 }
51                 dijkstra(s, n+nt, pos, adj, d_th, visit_th);
52                 for(index_t v = 0; v < n; v++)
53                     f_X[v] = d_th[v];
54             }
55         }
56         FREE(X_a);
57     }
58 }
```

Listing 6.6: A parallel implementation of the edge-linear algorithm for solving the Steiner problem in graphs.

parallelise the computations of $g_v(X)$ and $f_v(X)$ for all p-subsets of K for each $1 < p \leq k$.

Load balancing using default scheduling for all p-subsets of K for each $1 < p \leq k$ will result in poor parallel speed-up since the subsets are not uniformly distributed in the range 0 to $2^k - 1$. In our current implementation, we address this issue by pre-computing all the p-subsets of K for each $1 < p \leq k$ and apply default scheduling on these pre-computed subsets thereby distributing the load equally (see Lines 13–20 in Listing 6.6). More precisely, for each iteration with $1 < p \leq k$ we generate $\binom{k}{p}$ subsets and distribute it across c-cores with $\binom{k}{p}/c$ subsets on each core. However, we can avoid the pre-computations by using ranking and unranking functions [84] thereby generating the subsets more efficiently on-demand.

Let us recall from Section 5.5 that the edge-linear algorithm breaks down to finding 2^k shortest paths and hence 2^k executions of Dijkstra's algorithm. Essentially, this forms the core of the algorithm where essentially most computations are required. For each subset $X \subseteq K$ we invoke one Dijkstra procedure call to compute shortest paths from a single-source vertex to all other vertices in the graph. As a consequence of parallelisation over subsets, each core invokes an independent Dijkstra procedure call thereby computing shortest paths independently in parallel. Even though the memory accesses are arbitrary, we better saturate the memory bandwidth by making enough memory requests thereby keeping the pipeline busy.

In the next chapter, we will perform extensive experiments to check the scalability of our implementation of the edge-linear algorithm with respect to its runtime, memory-bandwidth and peak-memory usage.

Chapter 7

Experimental results

In this chapter, we present the experimental results of our implementation of the Erickson–Monma–Veinott algorithm for solving the Steiner problem in graphs. In addition, we report the scaling of Dijkstra's algorithm and an experimental comparison of the performance of binary and Fibonacci heaps with respect to Dijkstra's algorithm. To thoroughly examine the performance of our implementation, we need to take into account the architecture details and peak-achievable bandwidth of the hardware. For this reason, we report the configurations and baseline benchmarks of the hardware used for testing. Recall that, we referred the Erickson–Monma–Veinott algorithm as the edge-linear algorithm.

With scaling to large graphs as our primary objective, we design our experiments to study the runtime, memory bandwidth and peak-memory usage of our implementation of the edge-linear algorithm with respect to its edge scaling, parallelisation speed-up and terminal scaling. Our last set of experiments studies the overhead of tracking a Steiner tree.

7.1 Hardware configurations

In this section, we report the configurations of the hardware used for our experiments.

Mid-memory configuration. Dell PowerEdge C4130 2 x 2.5 GHz Intel Xeon E5-2680v3 CPU (Haswell microarchitecture, 24 cores, 12 cores/CPU, no hyper-threading, 30 MiB L3 cache), 128 GiB of main memory (8 x 16 GiB DDR4-2133 Samsung M393A2G40DB0-CPB, ECC enabled). CentOS

7.3 operating system with Linux kernel version 3.10.0-514.10.2.el7.x86_64 and `gcc` version 4.8.5.

Large-memory configuration. Dell PowerEdge C4130 2 x 2.5 GHz Intel Xeon E5-2680v3 CPU (Haswell microarchitecture, 24 cores, 12 cores/CPU, no hyper-threading, 30 MiB L3 cache), 256 GiB of main memory (16 x 16 GiB DDR4-2133 Samsung M393A2G40DB0-CPB, ECC enabled). CentOS 7.3 operating system with Linux kernel version 3.10.0-514.10.2.el7.x86_64 and `gcc` version 4.8.5.

Huge-memory configuration. Dell R930 4 x 2.8 GHz Intel Xeon E7-8891v3 CPU (Haswell microarchitecture, 40 cores, 10 cores/CPU, no hyper-threading, 45 MiB L3 cache), 1536 GiB of main memory (96 x 16 GiB DDR4-2133 Samsung M393A2G40DB0-CPB, ECC enabled). CentOS 6.8 operating system with Linux kernel version 2.6.32-642.6.1.el6.x86_64 and `gcc` version 4.9.3.

7.2 Measuring resource usage

The memory architecture of modern Intel microprocessors is hierarchical and pipelined (Intel [69, 70]). Each core has access to a number of dedicated registers and dedicated L1, L2 caches. However, all cores share access to the L3 cache and main memory (RAM). When a new memory-read request is issued, the processor checks if the requested data is available in the registers, L1, L2 and L3 caches in this order; if not, it is fetched from the main memory and passed along the hierarchy to the registers. While measuring the memory bandwidth we should only consider the memory-read requests issued to the main memory along the pipeline. One way of achieving this is by ignoring the subsequent memory accesses to the same memory address. We will demonstrate this approach in the later parts of this section.

In Intel documentation (Intel [70, Appendix B]; Levinthal [89]), it is reported that Intel Xeon microprocessors have a dedicated performance monitoring unit (PMU) which can be used to track the resource usage. The microprocessor has an uncore PMU which consists of eight programmable counters and one fixed counter. The uncore programmable counters can be configured to investigate the performance of L3 cache and main memory accesses. Additionally, each execution core has a per-core PMU which consists of four programmable counters and three fixed counters. The programmable per-core counters can be configured to investigate the performance of L1 and L2 caches, stalls inside the processor core and the latency of memory accesses. To enable

the counter measurements, the per-core and uncore PMUs support a number of performance events which can be used to initiate the measurements. For example, the uncore PMU counters can be used to measure the memory-read and memory-write bandwidth of memory accesses to the main memory using UNC_IMC_NORMAL_READS.– and UNC_IMC_WRITES.FULL.– class of events, respectively. In one hand, a more finegrained resource-utilisation statistics of the CPU can be obtained using the PMU counters. On the other hand, a lot of engineering effort is required to obtain the resource usage data. Additionally, Intel provides performance analysis and optimisation software such as Intel VTune and Intel Extreme Tuning utility which internally make use of the PMU counters to report the performance statistics. In a way, use of these optimisation software reduce the engineering effort required to obtain the resource usage statistics. However, these software are only available with a subscription cost. For these reasons using performance counters and Intel optimisation software for measuring the resource usage is kept for future work.

For the purpose of this thesis, we employ an alternative approach for measuring the resource usage. The running time of the experiments is measured via OpenMP wall-clock time interface (`omp_get_wtime`) and the memory usage is tracked by using wrapper functions for C standard library memory allocation interface (`malloc` and `free`). In the subsequent paragraphs, we discuss an approach for measuring the memory bandwidth of our implementation of the Erickson–Monma–Veinott algorithm and it proceeds as follows: First, we begin with a simple example to demonstrate the process of measuring the memory bandwidth. Second, we continue our discussion for measuring the memory bandwidth of the Dijkstra subroutine. Finally, we discuss an approach used to measure the memory bandwidth of our implementation of the edge-linear algorithm.

Let us begin our discussion with a simple example to measure the memory bandwidth of sequential memory accesses. In this example, we measure the memory bandwidth of the hardware for reading 64-bit words from consecutive memory addresses and the source-code is available in Listing 7.1. Lines 2–6 measure the memory bandwidth of a single core. We invoke a single thread in one of the cores which reads an array of 64-bit words sequentially. Lines 9–22 measure the memory bandwidth of all cores. We invoke exactly one thread per core and the memory accesses are equally distributed among all the threads. More precisely, each thread performs sequential memory accesses to the array elements within the bounds specified by *start* and *stop* variables (Lines 13–16). The memory bandwidth is the ratio of size of the array in bytes to the wall-clock time.

```
1  // read consecutive words - serial
2  push_time();
3  for(index_t i = 0; i < array_size; i++)
4      sum += array[i];
5  time = pop_time();
6  serial_bandwidth = inGiB(array_size*sizeof(index_t))/(time/1000.0);
7
8  // read consecutive words - parallel
9  push_time();
10 #pragma omp parallel for
11 for(index_t t = 0; t < num_threads; t++) {
12     index_t tsum  = 0;
13     index_t start = (array_size/num_threads)*t;
14     index_t stop  = (array_size/num_threads)*(t+1)-1;
15     for(index_t i = start; i <= stop; i++)
16         tsum += array[i];
17     sums[t] = tsum;
18 }
19 for(index_t t = 0; t < nt; t++)
20     sum += tsum[t];
21 time = pop_time();
22 parallel_bandwidth = inGiB(array_size*sizeof(index_t))/(time/1000.0);
```

Listing 7.1: An example source-code for measuring the memory bandwidth of read consecutive words experiment.

We will now proceed to measuring the memory bandwidth of our implementation of Dijkstra's algorithm described in Section 6.3. Let us recall from Section 5.5 that the edge-linear algorithm executes 2^k instances of Dijkstra subroutine for computing shortest paths. Each instance of the Dijkstra procedure traverses all edges in the input graph. Hence, for a graph instance with n vertices and m edges, the procedure reads $2n + 4m$ words (memory used to store the graph). The initialisation phase of Dijkstra's algorithm access $2n$ words and the visit-and-label phase access $2m$ words for updating the distances. Additionally, our implementation of Dijkstra's algorithm uses binary or Fibonacci heap to keep track of the minimum distance vertex and the exact number of priority-queue operations executed varies for each invocation of the Dijkstra subroutine. Furthermore, the number of heap elements accessed in Decrease-key and Extract-min operations varies for each invocation.

To keep track of the exact number of heap elements accessed we use counters and these counter values will eventually be used to compute the total memory accessed by the heap. We illustrate this using a simple example for counting the number of heap elements accessed while performing Decrease-key operation in binary heap. The source-code is available in Listing 7.2. In each iteration of the while loop (Lines 3–12 in Listing 7.2), we perform one key comparison in Line 8 with a new heap element. As a consequence, one heap element is accessed in each iteration of the while loop and the probability of

subsequent memory accesses (in Lines 9 and 10) to the same heap element resulting in a cache-hit is high. Hence, we ignore the memory accesses in Lines 8 and 9 for computing the memory bandwidth. Let x_h be the total number of heap elements accessed then the total memory accessed by heap operations is computed as $x_h \cdot$ `sizeof(heap_element_t)` bytes. We employ a similar approach to keep track of the number of heap elements accessed in the priority-queue operations of binary and Fibonacci heaps.

In our current implementation, we use a structure variable to keep track of the number of heap elements accessed. In contrast, the performance of our implementation of binary and Fibonacci heaps could substantially benefit from tracking the number of heap elements accessed locally (using a local variable) and updating the structure variable subsequently at the end of the subroutine. However, we plan to address this implementation limitation in the future versions of this software.

```
1  void decrease_key(bheap_t *h, index_t item, index_t new_key) {
2      index_t i   = h->p[item];
3      while(i >= 2) {
4          index_t j      = i / 2;
5          bheap_node_t y = h->a[j];
6          h->key_comps++; // key comparisons
7          h->mem++; // heap-elements accessed
8          if(key >= y.key) break;
9          h->a[i]       = y;
10         h->p[y.value] = i;
11         i = j;
12     }
13     h->a[i].value = value;
14     h->a[i].key   = key;
15     h->p[value]   = i;
16 }
```

Listing 7.2: An implementation of DECREASE-KEY operation in binary heap.

Finally, let us discuss an approach used for measuring the memory bandwidth of an implementation of the Erickson–Monma–Veinott algorithm described in Section 6.4. For a subset X of the terminal set K and a vertex v, let $f_v(X)$ denote the length of a Steiner tree connecting all the vertices in $X \cup \{v\}$. To compute a Steiner tree for a given problem instance with n vertices and k terminals the inner-most loop executes $3^k \cdot n$ comparisons of $\min\{f_v(X), f_v(X') + f_v(X \setminus X')\}$ for all $X \subseteq K$ and $\emptyset \neq X' \subset X$ (Line 38 in Listing 6.6). For each comparison, the values of $f_v(X)$, $f_v(X')$ and $f_v(X \setminus X')$ will be fetched from the memory. Hence, the total memory accessed by the inner-most loop of the edge-linear algorithm is $3^{k+1} \cdot n$ words. For computing the memory bandwidth of the edge-linear algorithm, we consider the memory transactions of the inner-most loop and the memory transactions of 2^k executions of the Dijkstra subroutine.

To summarise, the reported memory bandwidth of the edge-linear algorithm is the memory-read data transfer rate of the experiments and it is the ratio of the total number of bytes read from the main memory to the wall-clock time of the experiment. The total number of bytes read in our implementation of the edge-linear algorithm is computed as $(3^{k+1} \cdot n + 2^k(4n + 6m)) \cdot$ `sizeof(index_t)` $+ x_h \cdot$ `sizeof(heap_element_t)` bytes, where x_h is the total number of heap elements accessed.

7.3 Baseline performance

In this section, we benchmark the hardware and report their baseline performance. The hardware configurations are reported in Section 7.1. The benchmarking software is the contribution of Petteri Kaski.

Benchmarking is one of the approaches used for assessing the performance characteristics of the computer hardware (Henessey and Patterson [61]). Saavedra [107]; Vieira and Madeira [121] have used the benchmarking approach to identify potential performance penalties in the hardware. Benchmarking typically involves measuring the arithmetic and memory bandwidth of the hardware. Measuring these baseline benchmark values is essential to engineer and optimise an algorithm to achieve peak bandwidth performance.

In our experiments, the memory bandwidth of the hardware is measured by operating on a four-gigabyte array of 64-bit words. We design four experiments which simulate the possible memory-access scenarios and the experiments are designed as follows: First, in *read consecutive words* experiment we perform sequential memory access, more precisely we read 64-bit words from linear memory addresses. Second, in *read random words* experiment we read individual 64-bit words from random memory addresses. Third, in *read random cache lines* experiment we read complete cache line from random memory addresses. Finally, in *write consecutive words* experiment we perform sequential writing, precisely we write 64-bit words to linear memory addresses. The baseline performance of the hardware is reported in Table 7.1 and the experiments bear the delay of generating random addresses.

Each experiment is executed five times and the average memory bandwidth of five iterations is reported. To avoid cold start, the hardware is warmed-up by executing one iteration of the experiment before we start measuring the memory bandwidth. Furthermore, the experiments are repeated for a single core and all cores of the test hardware. To parallelise across multiple execution cores, we use OpenMP API [100] via the `omp parallel for` construct with default scheduling. The random-memory addresses are gen-

erated using a pseudo-random generator from Björklund *et al.* [11] and it is the exclusive-or of the two states of two 64-bit linear-feedback shift registers (LFSRs). One LFSR state is shifted towards the more significant bits and another LFSR state is shifted towards less significant bits. The states of both LFSRs can be fast-forwarded with square-and-multiply exponentiation in the polynomial quotient ring defined by a tap polynomial.

Benchmark	Single core	All cores
Mid-memory configuration		
Read consecutive words	8.35 GiB/s	36.22 GiB/s
Read random words	0.53 GiB/s	2.50 GiB/s
Read random cache lines	1.54 GiB/s	7.73 GiB/s
Write consecutive words	6.37 GiB/s	24.33 GiB/s
Large-memory configuration		
Read consecutive words	8.43 GiB/s	39.93 GiB/s
Read random words	0.56 GiB/s	2.45 GiB/s
Read random cache lines	1.52 GiB/s	7.91 GiB/s
Write consecutive words	6.23 GiB/s	26.86 GiB/s
Huge-memory configuration		
Read consecutive words	3.99 GiB/s	36.64 GiB/s
Read random words	0.19 GiB/s	4.63 GiB/s
Read random cache lines	0.58 GiB/s	16.83 GiB/s
Write consecutive words	3.24 GiB/s	22.16 GiB/s

Table 7.1: The baseline performance of the test hardware. A word is 64-bits and a cache line is eight consecutive words or a group of 512-bits.

7.4 Input graphs

To test the correctness of our implementation, we use the instances of the Steiner problem from Koch, Martin and Voß [83]. As our implementation mainly focuses on scaling with small values of k, we ignore all test instances with $k > 20$.

For scalability testing we generate synthetic graphs using a random graph generator from Björklund *et al.* [11]. For nonnegative integer values d, n with dn even, the generator uses the configuration model from Bollobás [12, Section 2.4] to create a d-regular, n-vertex random graph with nonnegative integer weights associated with the edges and the edge weights are assigned

randomly in the range 1 to 2^{32}. The generator does not reject the configurations with loops (1-cycles) or couplings (2-cycles). The generated graphs are in DIMACS STP format [83].

To achieve consistency in results we generate identical graphs across all machines. More precisely, given the number of vertices n, degree d of each vertex, number of terminals k and a *seed* value, the generator produces same graph instance even with different C libraries and different test environments.

7.5 Edge-scaling of Dijkstra's algorithm

Our first set of experiments study the scaling of our implementation of Dijkstra's algorithm described in Section 6.3. We measure the runtime, memory bandwidth and peak-memory usage of the algorithm as a function of the number of edges m. In addition, we report the performance evaluation of binary and Fibonacci heaps in Dijkstra's algorithm. We use our implementation of binary and Fibonacci heaps described in Section 6.2. The experiments are performed on a single core of the mid-memory configuration and the results are displayed in Figure 7.3.

We observe that the runtime and memory usage of the algorithm increase linearly as we increase m and the runtime has little variance between the independent graph inputs. One of the important observations is the decrease in memory bandwidth with increase in number of edges m. The decrease in memory bandwidth could be the result of cache misses with increase in m. For input graphs with small number of edges, the graph completely fits inside the L3 cache of the mid-memory configuration. As we increase the graph size, the L3 cache will not be sufficient to store the entire graph and a part of the graph is stored in the main memory. Recall from Section 6.1 that each execution of Dijkstra procedure traverses all the edges in the graph, which results in an arbitrary pattern of memory accesses to the main memory and the data access patterns will essentially have less cache locality. As a consequence of this, we pay the penalty for cache misses.

Let us recall from Chapter 4 that, we discussed the advances in priority-queue implementations and the performance of Dijkstra's algorithm relies on the asymptotic complexity of the priority-queue operations. Indeed, Dijkstra's algorithm using Fibonacci heap has better theoretical performance than the binary heap (see Section 4.8). However, from our experimental results it is observed that the performance of our implementations of the binary heap is better than Fibonacci heap with respect to finding a shortest path using Dijkstra's algorithm. We investigate this in more detail in the next section.

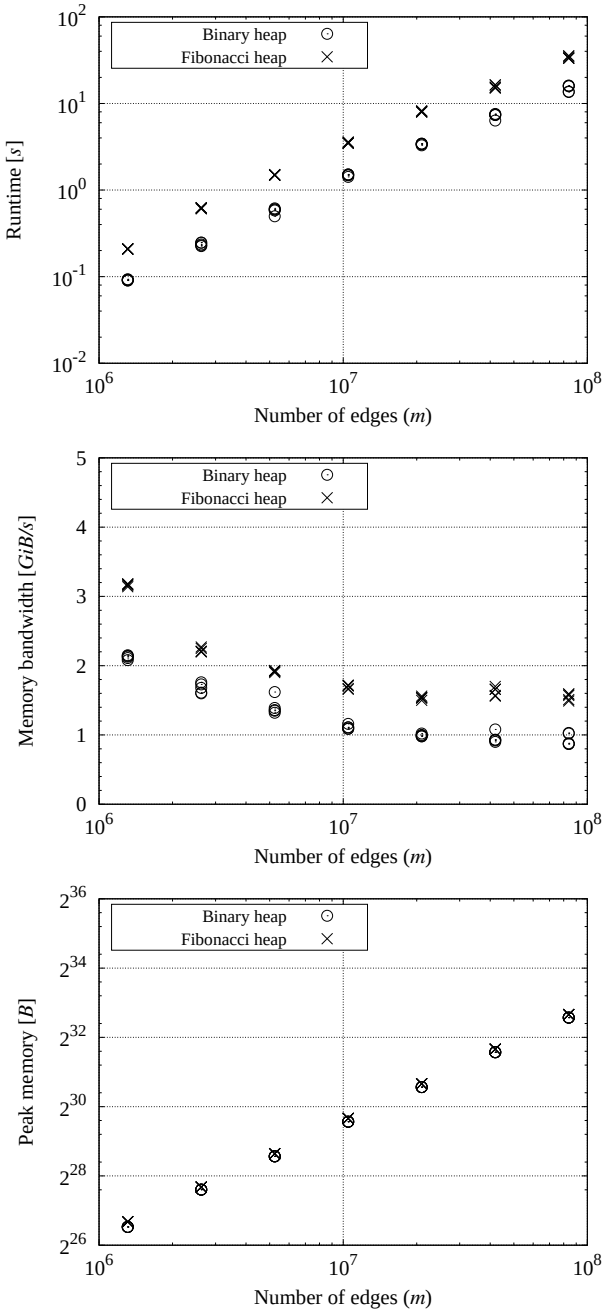


Figure 7.3: Performance evaluation of binary and Fibonacci heaps in edge scaling of Dijkstra's algorithm. We display the runtime (top), memory bandwidth (center) and memory usage (bottom) of Dijkstra's algorithm for five independent d-regular random graphs for each $n = 2^{17}, 2^{18}, \ldots, 2^{23}$, $d = 20$ fixed and a random source vertex. The experiments are performed separately for binary and Fibonacci heaps on a single core of the mid-memory configuration. Note that the horizontal axis is logarithmic and the vertical axis of the runtime (top) and memory usage (bottom) plots is also logarithmic.

7.6 Binary heaps versus Fibonacci heaps

The Fibonacci heap supports DECREASE-KEY and INSERT priority-queue operations in amortised constant time compared to $O(\log n)$ time in the binary heap. From an implementation perspective, the decrease in asymptotic complexity of the INSERT operation is essentially inconsequential as it does not affect the runtime of Dijkstra's algorithm significantly. However, we can potentially benefit from the improved time complexity of the DECREASE-KEY operation if it is used in an application where the DECREASE-KEY operations are frequent (Cormen *et al.* [24, Chapter 19]). In terms of Dijkstra's algorithm this means that the underlying graph should be dense. We design our second set of experiments to study the effect of graph density on the performance of our implementation of binary and Fibonacci heaps in Dijkstra's algorithm, and report the runtime of the experiments in Figures 7.4 and 7.5. Our implementation of the binary and Fibonacci heaps is described in Section 6.2.

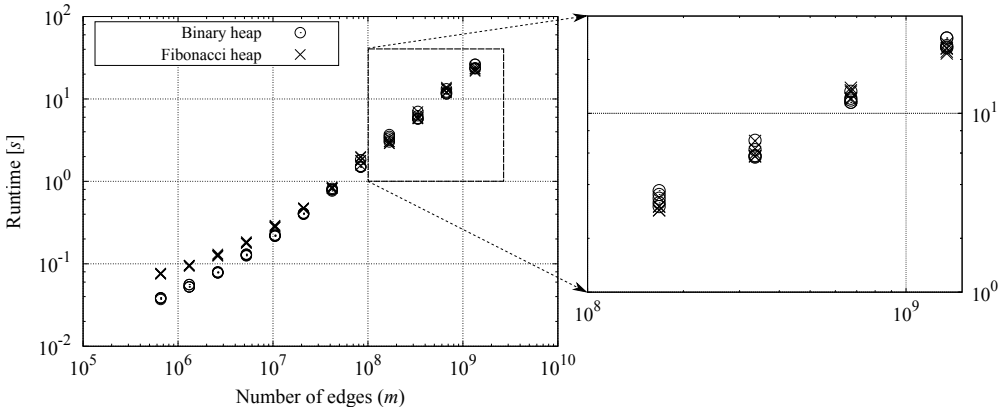


Figure 7.4: Performance evaluation of the binary and Fibonacci heaps with respect to the graph density. We display the runtime of Dijkstra's algorithm for five independent d-regular random graphs for each $d = 2 \cdot 10, 2^2 \cdot 10, \ldots, 2^{12} \cdot 10$ with $n = 65536$ fixed and a random source vertex. The experiments are executed on a single core of the mid-memory configuration. All axes have logarithmic scale. The zoom-out section is displayed for better visibility and the runtime of binary heap is slowly overtaking the runtime of Fibonacci heap for dense graphs.

From theoretical perspective, we expect Fibonacci heap to perform better for dense graphs because of frequent DECREASE-KEY operations. However, in practice even though there is a significant improvement in the performance with increase in the graph density, it does not outperform the binary heap.

One possible explanation could be due to the large constant factors in the Fibonacci heap operations. Our implementation of Fibonacci heap uses an explicit and memory-consuming pointer structure. Each elementary operation in Fibonacci heap involve the storage manipulation of four pointers per element and it is an additional overhead compared to the binary heap which only stores the priority and value of each element. Even though we observe a significant improvement in the memory bandwidth using Fibonacci heap, we pay the penalty of large constant factors in the Fibonacci heap operations.

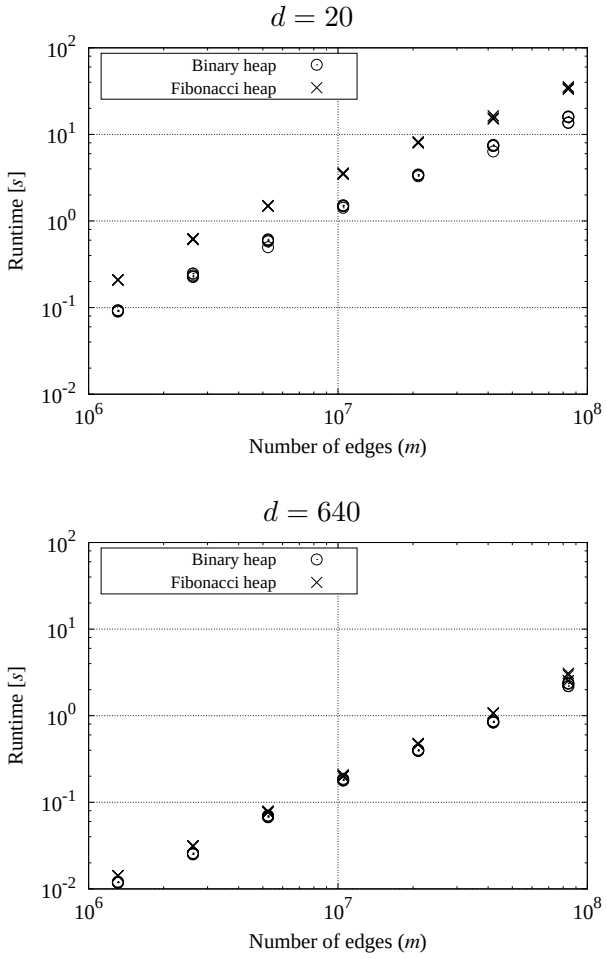


Figure 7.5: Performance comparison of the binary and Fibonacci heaps in Dijkstra's algorithm. We display the runtime of Dijkstra's algorithm for five independent d-regular random graphs for each $n = 2^{17}, 2^{18}, \ldots, 2^{23}$, $d = 20$ fixed (top) and for each $n = 2^{12}, 2^{13}, \ldots, 2^{18}$ with $d = 640$ fixed (bottom) and a random source vertex. The experiments are performed on a single core of the mid-memory configuration. All axes have logarithmic scale.

7.7 Scaling of the edge-linear algorithm

In this section, we discuss the experimental results of our implementation of the Erickson–Monma–Veinott algorithm for the Steiner problem in graphs described in Section 6.4. Recall that, we refer the Erickson–Monma–Veinott algorithm as the edge-linear time algorithm. For all the experiments performed in this section, we employ our implementation of the binary heap instead of the Fibonacci heap, since the former is more efficient based on the experiments in Section 7.6.

Edge-linear scaling. Our third set of experiments study the scaling of the edge-linear algorithm for the Steiner problem in graphs. More specifically, we study the runtime, memory bandwidth and peak-memory usage of the algorithm as we increase the number of edges m. The results of the experiments are displayed in Figure 7.6. The experiments are performed on a single core of the mid-memory configuration. We observe that the runtime and memory usage of the algorithm increase linearly as we increase m and the runtime has little variance between the independent graph inputs. On the other hand, we observe a decrease in the memory bandwidth with increase in m. Recall that the edge-linear algorithm executes 2^k iterations of the Dijkstra subroutine for computing shortest paths. A potential explanation for the decrease in memory bandwidth of the edge-linear algorithm could be a consequence of the decrease in memory bandwidth of our implementation of Dijkstra's algorithm with increase in m (see Section 7.5).

Parallelisation speed-up. The next set of experiments study the performance improvements in the edge-linear algorithm with parallelisation. Specifically, we compare the change in runtime, memory bandwidth and peak-memory usage of the algorithm as a function of the number of edges m. In addition, we report the runtime and memory bandwidth comparisons in Table 7.2 and Table 7.3, respectively. The experiments are performed on the mid-memory configuration and the results are displayed in Figure 7.7. From the experimental results, we observe linear increase in the runtime and memory usage, and decrease in the memory bandwidth as we increase m in both serial and parallel variants of the software implementation. From our preliminary analysis, we observe that the performance of the implementation appears to be constrained by the memory bandwidth of the underlying hardware and the speed-up in runtime directly corresponds to the bandwidth ratio. (See Tables 7.2 and 7.3.)

Scaling up to a billion edges. Our experiments in this section study the scaling of the edge-linear algorithm up to a billion edges with respect to its

runtime, memory bandwidth and peak-memory usage as a function of the number of edges m. The experiments are performed on the huge-memory configuration and the results of the experiments are displayed in Figure 7.8. We observe linear scaling of the runtime and memory usage with little variance among the independent graph inputs. The decrease in memory bandwidth could be the result of cache misses with the increase in graph size. It is important to remember that the algorithm requires memory resources that grow exponentially in the number of terminals. For a graph with a billion edges and ten terminals, the algorithm requires approximately a terabyte of working memory.

Exponential scaling in the number of terminals. Our final set of experiments in this section study the scalability of the edge-linear algorithm as a function of the number of terminals k. We study the runtime, memory bandwidth and peak-memory usage of the algorithm as we increase k. The experiments are performed on the mid-memory configuration. The results of the experiments are displayed in Figure 7.9. The runtime and memory usage of the algorithm increases exponentially with increase in k. An important observation is the significant increase in the memory bandwidth. The increase in throughput could be a consequence of cache locality of the graph. We use graphs with small number of edges and the graph completely fits inside the L3 cache of the mid-memory configuration which potentially improves the cache locality.

n	Serial	Parallel	Speed-up
2^{17}	79.43 s	10.63 s	7.48
2^{18}	175.75 s	25.01 s	7.03
2^{19}	456.80 s	59.69 s	7.65
2^{20}	1139.75 s	133.78 s	8.52
2^{21}	2609.16 s	287.63 s	9.07
2^{22}	7496.59 s	600.91 s	12.48
2^{23}	18979.55 s	1240.08 s	15.31

Table 7.2: Runtime comparison of serial and parallel implementations of the edge-linear algorithm. We report the average runtime of the algorithm on a single core and all cores of the mid-memory configuration for five independent d-regular random graphs for each $n = 2^{17}, 2^{18}, \ldots, 2^{23}$, $d = 20$ fixed and $k = 10$ fixed. The *speed-up* is the ratio of the serial and parallel runtime. All reported times are in seconds.

n	Serial	Parallel	Bandwidth ratio
2^{17}	3.55 GiB/s	26.45 GiB/s	7.46
2^{18}	3.29 GiB/s	20.51 GiB/s	6.24
2^{19}	2.54 GiB/s	19.45 GiB/s	7.65
2^{20}	2.07 GiB/s	17.61 GiB/s	8.53
2^{21}	1.83 GiB/s	16.60 GiB/s	9.07
2^{22}	1.30 GiB/s	16.11 GiB/s	12.43
2^{23}	1.06 GiB/s	15.86 GiB/s	14.99

Table 7.3: Memory-bandwidth comparison of serial and parallel implementations of the edge-linear algorithm. We report the average memory bandwidth of the algorithm on a single core and all cores of the mid-memory configuration for five independent d-regular random graphs for each $n = 2^{17}, 2^{18}, \ldots, 2^{23}$, $d = 20$ fixed and $k = 10$ fixed. The *bandwidth ratio* is the ratio of the memory bandwidth of the parallel and serial implementations.

7.8 Optimal cost versus optimal solution

In our previous experiments, we only studied the performance of the edge-linear algorithm for computing the cost of a Steiner tree. In this section, we design a set of experiments to study the overhead of tracking a Steiner tree. To be precise, we compare the runtime, memory bandwidth and peak-memory usage of the optimal-cost and optimal-solution variants of the software implementation as we increase the number of edges m. Tracking a Steiner tree does not involve additional memory-read operations. However, it essentially requires additional memory-write operations to store the bookkeeping information (Lines 4, 13, 32 in Algorithm 3).

We display the experimental comparison of the optimal-cost and optimal-solution variants of the software implementation of the edge-linear time algorithm in Figure 7.10. The experiments are performed on the large-memory configuration and we use binary heap. The runtime and memory usage grows linearly with increase in m and the runtime has little variance between independent graph inputs in both variants of the software. The difference in runtime between the optimal-cost and optimal-solution implementations is small. However, tracking the optimal-solution requires significantly more memory resources and it is approximately three times the peak-memory usage of the optimal-cost variant.

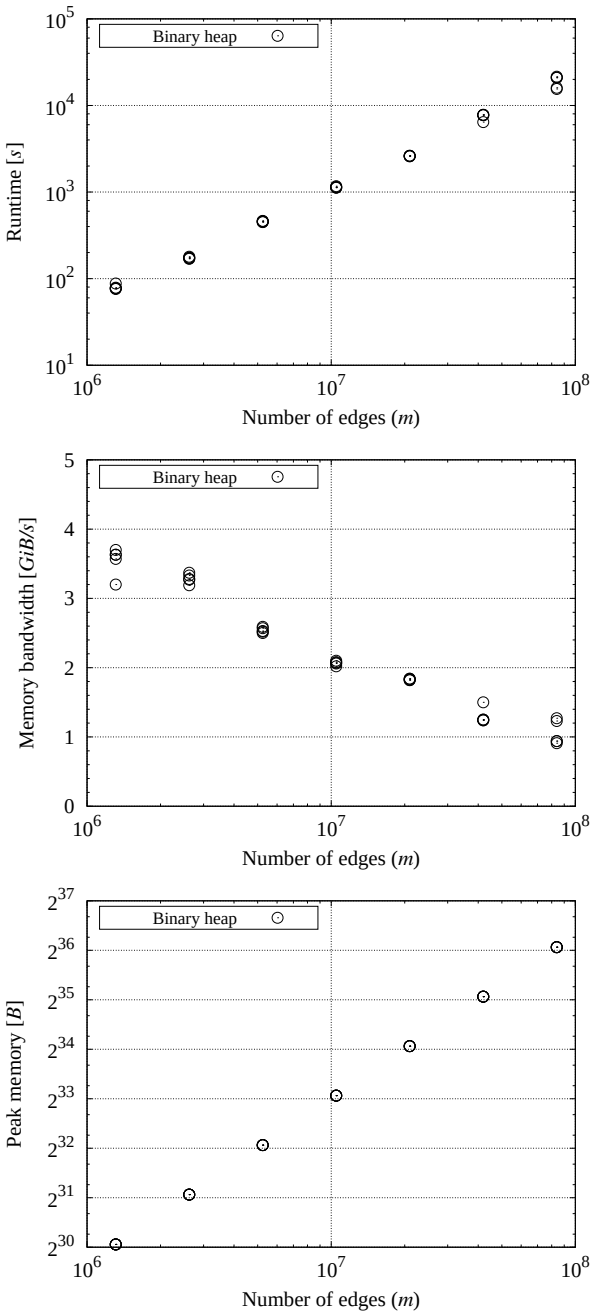


Figure 7.6: Scalability of the edge-linear time algorithm as we increase the number of edges m. We display the runtime (top), memory bandwidth (center) and peak-memory usage (bottom) of the edge-linear algorithm for five independent d-regular random graphs for each $n = 2^{17}, 2^{18}, \cdots, 2^{23}$, $d = 20$ fixed and $k = 10$ fixed. The experiments are performed on a single core of the mid-memory configuration. Note that the horizontal axis is logarithmic and the vertical axis of runtime plot (top) and memory-usage plot (bottom) is also logarithmic. We use binary heap as the priority queue.

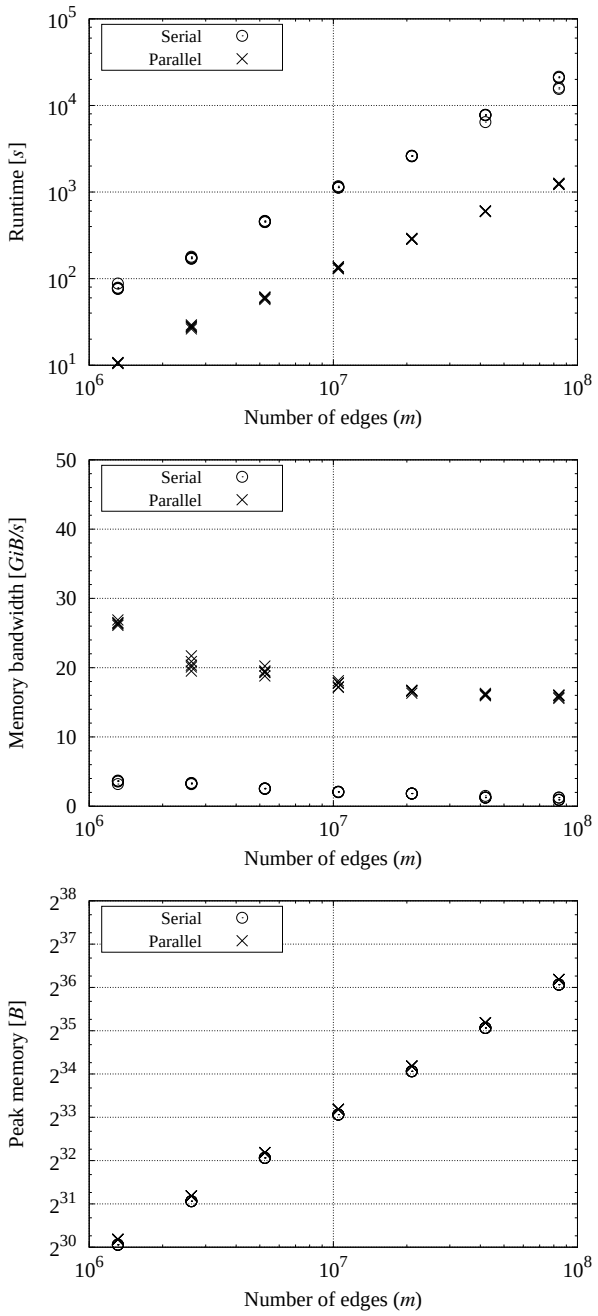


Figure 7.7: Performance comparison of serial and parallel implementations of the edge-linear time algorithm as a function of the number of edges m on the mid-memory configuration. We display the runtime (top), memory bandwidth (center) and peak-memory usage (bottom) of five independent d-regular random graphs for each $n = 2^{17}, 2^{18}, \ldots, 2^{23}$, $d = 20$ fixed and $k = 10$ fixed. The horizontal axis is logarithmic and the vertical axis of the runtime plot (top) and memory-usage plot (bottom) is also logarithmic. We use binary heap instead of Fibonacci heap.

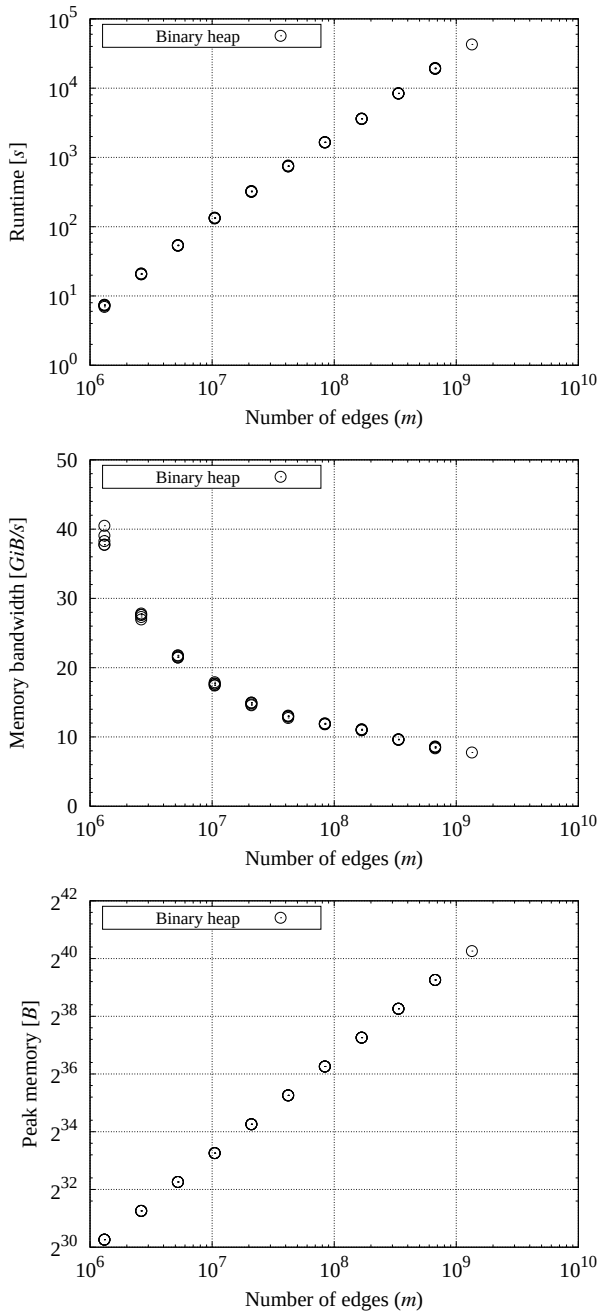


Figure 7.8: Scalability of our parallel implementation of the edge-linear time algorithm up to a billion edges. We display the runtime (top), memory bandwidth (center) and peak-memory usage (bottom) of the algorithm for five independent d-regular random graphs for each $n = 2^{17}, 2^{18}, \cdots, 2^{27}$, $d = 20$ fixed and $k = 10$ fixed. The experiments are performed on the huge-memory configuration using the binary heap. Note that the horizontal axis is logarithmic, and the vertical axis of runtime plot (top) and memory usage plot (bottom) is also logarithmic.

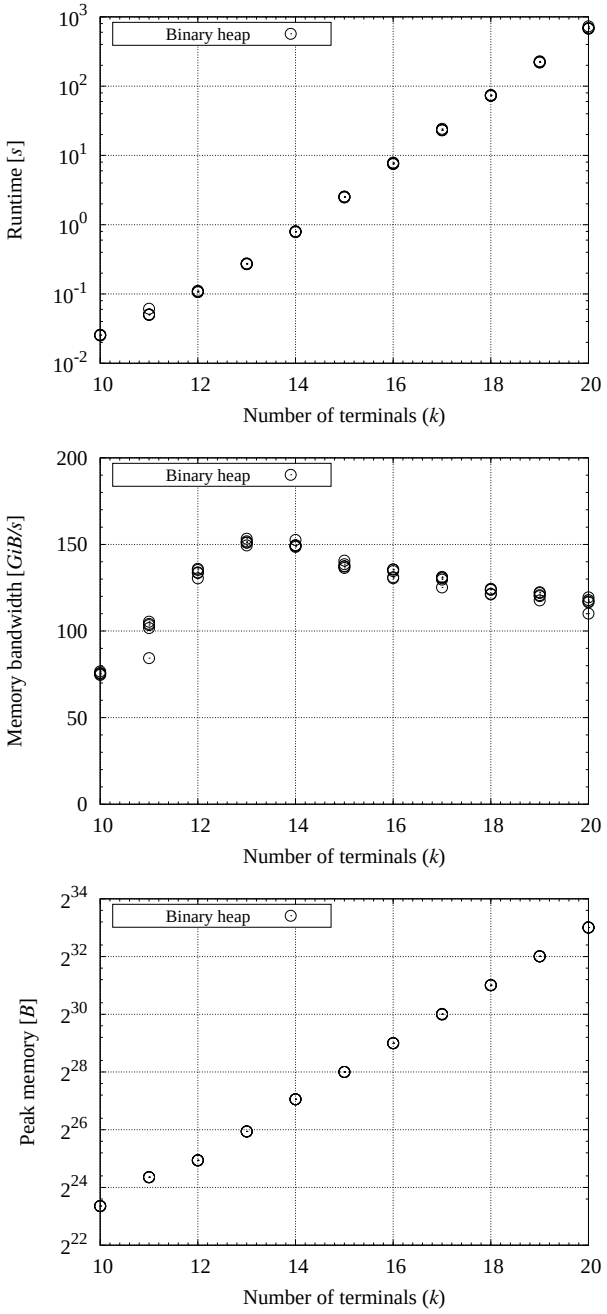


Figure 7.9: Exponential scaling in the number of terminals for our parallel implementation of the edge-linear algorithm. We display the runtime (top), memory bandwidth (middle) and peak-memory usage (bottom) of the algorithm for five independent d-regular random graphs for each $k = 10, 11, \ldots, 20$, $n = 1024$ fixed and $d = 20$ fixed. The experiments are performed on the mid-memory configuration. The vertical axis of the runtime plot (top) and memory usage plot (bottom) is logarithmic. We use binary heap as the priority queue.

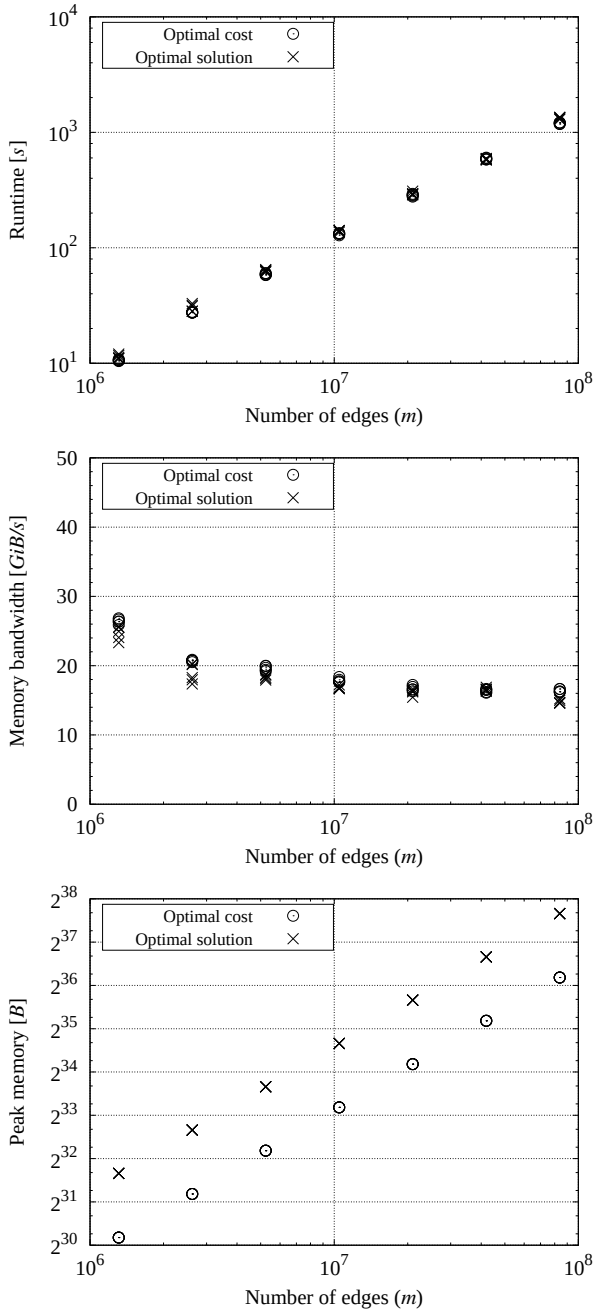


Figure 7.10: Performance comparison of optimal cost versus optimal solution implementations of the edge-linear time algorithm. Both implementations are parallel variants and they make use of all the available execution cores in the test hardware. We display the runtime (top), memory bandwidth (center) and peak-memory usage (bottom) of five independent d-regular random graphs with $n = 2^{17}, 2^{18}, \ldots, 2^{23}$, $d = 20$ fixed and $k = 10$ fixed. The experiments are performed on the large-memory configuration using the binary heap. The horizontal axis is logarithmic and the vertical axis of the runtime plot (top) and memory usage plot (bottom) is also logarithmic.

7.9 Performance improvements

The memory usage of the algorithm can be reduced with further implementation engineering. Our implementation of the edge-linear algorithm uses 64-bit indexing for subset representation which can be reduced to 32 bits. In our implementation of Dijkstra's algorithm, we insert all elements in to the heap during the initialisation phase and update the distances using DECREASE-KEY operation in the visit-and-label phase (see Section 6.3); however an alternative implementation can start with an empty heap and new elements can be inserted as they are discovered. Even though the alternative variant has the same worst-case bounds, it reduces the memory usage by maintaining a smaller priority queue and practically speeds up the priority-queue operations. A combination of speed-up techniques have been proposed to improve the space-time trade-off of Dijkstra's algorithm (Kumar and Schwabe [85]; Meyer and Zeh [95]). However, employing these speed-up techniques is left for future work.

An additional property of our implementation is that, it can compute a Steiner tree for a new terminal set with one additional terminal along with the original terminal set using the bookkeeping information without any extra computations. However, the number of computations required to compute a Steiner tree explicit to the given terminal set K can be reduced by half. More precisely, we can compute the cost of a Steiner tree for the subset $K \setminus \{v\}$ for some $v \in K$ and extend the set as $K \setminus \{v\} \cup \{v\}$ for computing the cost of a Steiner tree for K. By doing this, the runtime of experiments for computing a Steiner tree exclusive to the given problem instance can be reduced by half.

Chapter 8

Conclusion

A large number of real-world problems and phenomena can be modelled using graphs; in recent years, the graphs modelled from the real-world problems have grown in size. For example, consider the problem of finding a group of participants to organise an event in Facebook network [5] with approximately 700 million active users and 70 billion friendship links. The success of the event will be higher if we invite a set of well-acquainted friends along with other participants. Sozio and Gionis [113] have demonstrated that such a problem can be modelled as the Steiner problem in graphs. The problem of finding a group of individuals in a social network who can function as a team to accomplish a particular task has been modelled as the group Steiner problem by Lappas, Liu and Terzi [88]. In addition, a large number of communication and infrastructure network planning, social and graph mining problems can be reduced to the problem of finding the Steiner and group Steiner trees. Our implementation of the edge-linear algorithm for the Steiner problem has potential applications for finding a Steiner tree in large graph instances.

The Steiner problem is one of the twenty-one original $\mathcal{NP}$-complete problems discussed by Karp [77]. From the algorithm design perspective, the Steiner problem exhibits both formidable hardness and ease of scalability, meaning that even though the problem is $\mathcal{NP}$-complete it admits algorithms that run in polynomial time in the size of the input graph and the exponential complexity can be restricted to the number of terminals. Many parameterised algorithms have been developed to solve the Steiner problem in graphs including Cygan *et al.* [26]; Downey and Fellows [33]; Erickson, Monma and Veinott [38]; Flum and Grohe [42], Fomin *et al.* [44]; Fomin *et al.* [45]. To design efficient algorithms for the Steiner problem, a sophisticated mathematical theory has been developed around the problem building on a combination

of techniques including combinatorics, graph theory and algebra. We have presented a review of state-of-the-art exact and approximation algorithms for solving the Steiner and group Steiner problems in Chapter 5.

In this thesis, we discussed two parameterised algorithms for solving the Steiner problem, and by reduction, the group Steiner problem: (i) a dynamic-programming algorithm presented by Dreyfus and Wagner [34]; and (ii) an improvement of the Dreyfus–Wagner algorithm presented by Erickson, Monma and Veinott [38] that runs in linear time in the size of the input graph and the exponential complexity is restricted to the number of terminals. We referred the Erickson–Monma–Veinott algorithm for solving the Steiner problem as the edge-linear algorithm. The dynamic-programming idea of Dreyfus and Wagner has been used extensively by many researchers and computer scientists including Björklund *et al.* [10]; Erickson, Monma and Veinott [38]; Fuchs *et al.* [48]; Fuchs *et al.* [49]; Hougardy, Silvanus and Vygen [63] to design many Steiner tree algorithms with fast theoretical worst-case behaviour. Additionally, we discussed a linear-time reduction presented by Voß [122] for transforming the group Steiner problem to the Steiner problem. Using this transformation, most algorithms for solving the Steiner problem can be used to solve the group Steiner problem.

As a primary objective of this thesis, we presented a scalable implementation of the edge-linear algorithm for solving the Steiner problem in graphs, and successfully parallelised the implementation to achieve speed-up in computations. An additional feature of our implementation is that it can compute a Steiner tree for a terminal set with an additional vertex along with the original terminal set using the bookkeeping information without any extra computations. To thoroughly examine the performance of the algorithm implementation, we presented the baseline benchmarks of the hardware. From our benchmarking results it is observed that a single core is not sufficient to saturate the memory interface and the peak-achievable throughput is less than the peak throughput specified by the hardware vendor. To fully saturate the memory interface, there should be sufficient number of memory-read and memory-write requests executed in each execution core (see Section 7.3).

In the previous chapter, we studied the behaviour of our implementation of the edge-linear algorithm with respect to its runtime, memory bandwidth and peak-memory usage. Our experimental results have demonstrated that the implementation can scale up to a billion edges, provided that the number of terminals is small. As the graph size increases, it can easily outgrow the computation and memory capacities of a single core. To achieve speed-up in computations we parallelised the edge-linear algorithm across all cores of

a multi-core CPU. In the edge-linear algorithm, the cost of a Steiner tree connecting a p-subset of the terminal set depends only on the subsets with size less than p. Hence, we parallelised the computations of finding the cost of the Steiner trees for all the p-subsets of the terminals set (see Section 6.4). For a graph instance with one-hundred million edges and ten terminals, our parallel implementation of the edge-linear algorithm is fifteen times faster than its serial counterpart on a Haswell compute node with two processors and twelve cores in each processor (see Table 7.3). Additionally, we observed that the memory bandwidth of our parallel implementation is at least twice the bandwidth of read random cache lines experiment for large graphs up to hundred million edges (see Tables 7.1 and 7.3). For graphs with small number of edges our implementation can scale up to twenty terminal vertices.

The edge-linear algorithm uses Dijkstra's algorithm as a subroutine for computing a shortest path. Our implementation of Dijkstra subroutine uses priority-queue implementations such as binary and Fibonacci heaps to keep track of the minimum distance vertex. Consequently, the performance of the Dijkstra subroutine relies on the efficiency of the priority-queue operations. We presented an experimental comparison of the performance of our implementation of binary and Fibonacci heaps in Chapter 7. Contrary to theoretical expectations, the binary heap performs better than the Fibonacci heap across our range of experiments for finding a shortest path using Dijkstra's algorithm. Nevertheless, Fibonacci heap can compete with binary heap provided that the underlying graph is large and dense; but it does not outperform the binary heap (see Section 7.6). From an algorithm engineering perspective, it is sensible to use the binary heap considering the implementation effort required and no practical speedup achieved. Additionally, we achieved significantly high memory bandwidth in edge scaling of Dijkstra's algorithm for dense graphs. The improved memory bandwidth for dense graphs is a consequence of memory prefetching and improved cache locality of the array of arrays representation of graphs. Our experimental results are in correspondence with the results reported of Lumsdaine *et al.* [92]. They studied the performance of the array of arrays representation of the graphs with respect to Dijkstra's algorithm.

In conclusion, this work has presented a scalable implementation of the edge-linear algorithm for solving the Steiner and group Steiner problems in graphs. Our implementation has been shown to be easily parallelisable allowing the algorithm to achieve moderate memory bandwidth. The experimental results show that the edge-linear algorithm can scale up to a billion edges. From our preliminary analysis, we observed that the performance of our implementation is limited by the memory bandwidth of the hardware (see

Tables 7.2 and 7.3), and we only utilise forty percent of the peak-achievable bandwidth for large graph instances. This shows that there most likely still is some room for improvement with respect to achieving higher memory bandwidth. A possible approach for improving the memory bandwidth include designing memory-hierarchy-sensitive graph layouts described by Furaih and Ranka [50], and Wei *et al.* [127] in the context of improving the performance of Dijkstra's algorithm. Additionally, many alternative approaches have been suggested to improve the performance of Dijkstra's algorithm using parallel priority queues such as relaxed heaps [35] and parallel heaps [15]. However, employing these approaches is left for future work.

Using graphical processing units (GPUs) for other uses than their original purpose is becoming more common. Algorithm implementations can often benefit from being implemented on GPUs especially if the implementations are sufficiently parallelisable. Modern GPUs provides hundreds of gigabytes of arithmetic and memory bandwidth. For example, GPU accelerators such as NVIDIA K80 with two Tesla GK210 GPUs [98] and NVIDIA P100 with one Pascal GP100 GPU [99] possess 480 GiB/s and 732 GiB/s of theoretical memory bandwidth, respectively. These modern GPUs could be a potential platform for implementing the edge-linear algorithm. However, GPU microarchitectures are optimised for coalescent and vectorised execution. This means that to achieve peak memory and arithmetic bandwidth, the algorithm design should support coalescent execution and coalesced memory accesses at least in warp level. On the contrary, the edge-linear algorithm exhibits divergent memory accesses, which makes it highly unlikely to achieve peak-memory bandwidth. Nevertheless, even by utilising only one fourth of the peak-achievable bandwidth of the GPUs we can achieve a significant speed-up in computations.

Bibliography

[1] Ahuja, R. K., Mehlhorn, K., Orlin, J., and Tarjan, R. E. Faster algorithms for the shortest path problem. *Journal of the ACM 37*, 2 (1990), 213–223.

[2] Anagnostopoulos, A., Becchetti, L., Castillo, C., Gionis, A., and Leonardi, S. Online team formation in social networks. In *Proceedings of the Twenty-first International Conference on World Wide Web* (2012), ACM, pages 839–848.

[3] Anderson, S. E. Bit twiddling hacks. webpage. `https://graphics.stanford.edu/~seander/bithacks.html`. Accessed 15.07.2017.

[4] Arora, S., and Barak, B. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[5] Backstrom, L., Boldi, P., Rosa, M., Ugander, J., and Vigna, S. Four degrees of separation. In *Proceedings of the Fourth Annual ACM Web Science Conference* (2012), ACM, pages 33–42.

[6] Bateman, C. D., Helvig, C. S., Robins, G., and Zelikovsky, A. Provably good routing tree construction with multi-port terminals. In *Proceedings of the ACM SIGDA International Symposium on Physical Design* (1997), ACM, pages 96–102.

[7] Bellman, R. On a routing problem. *Quarterly of Applied Mathematics 16* (1958), 87–90.

[8] Berman, P., and Ramaiyer, V. Improved approximations for the Steiner tree problem. *Journal of the Algorithms 17*, 3 (1994), 381–408.

[9] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., and Sudarshan, S. Keyword searching and browsing in databases using banks. In *Proceedings of the Eighteenth International Conference on Data Engineering* (2002), IEEE, pages 431–440.

[10] Björklund, A., Husfeldt, T., Kaski, P., and Koivisto, M. Fourier meets Möbius: fast subset convolution. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing* (2007), ACM, pages 67–74.

[11] Björklund, A., Kaski, P., Kowalik, L., and Lauri, J. Engineering motif search for large graphs. In *Proceedings of the Seventeenth workshop on Algorithm Engineering and Experiments* (2015), SIAM, pages 104–118.

[12] Bollobàs, B. *Random Graphs*, second edition, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 2001.

[13] Brazil, M., Graham, R. L., Thomas, D. A., and Zachariasen, M. On the history of the Euclidean Steiner tree problem. *Archive for History of Exact Sciences 68*, 3 (2014), 327–354.

[14] Brodal, G. S., Lagogiannis, G., and Tarjan, R. E. Strict Fibonacci heaps. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing* (2012), ACM, pages 1177–1184.

[15] Brodal, G. S., Träff, J. L., and Zaroliagis, C. D. A parallel priority data structure with applications. In *Proceedings of the Eleventh International Symposium on Parallel Processing* (1997), IEEE, pages 689–693.

[16] Brown, M. R. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing 7*, 3 (1978), 298–319.

[17] Brown, M. R., and Tarjan, R. E. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing 9*, 3 (1980), 594–614.

[18] Byrka, J., Grandoni, F., Rothvoss, T., and Sanità, L. An improved lp-based approximation for Steiner tree. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing* (2010), ACM, pages 583–592.

[19] Charters, B. A. Algorithm 311: Prime number generator 2. *Communications of the ACM 10*, 9 (1967), 570.

[20] Chiang, M., Lam, H., Liu, Z., and Poor, H. V. Why steiner-tree type algorithms work for community detection. In *Proceedings*

of the Sixteenth International Conference on Artificial Intelligence and Statistics (2013), JMLR, pages 187–195.

[21] Chlebík, M., and Chlebíková, J. The Steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science 406*, 3 (2008), 207–214.

[22] Cockayne, E. J., and Melzak, Z. A. Euclidean constructibility in graph-minimization problems. *Mathematics Magazine 42*, 4 (1969), 206–208.

[23] Coffman, J., and Weaver, A. C. An empirical performance evaluation of relational keyword search techniques. *IEEE Transactions on Knowledge and Data Engineering 26*, 1 (2014), 30–42.

[24] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, third edition. MIT Press, 2009.

[25] Courant, R., and Robbins, H. *What is Mathematics? An Elementary Approach to Ideas and Methods*. Oxford University Press, 1941.

[26] Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., and Saurabh, S. *Parameterized Algorithms*, volume 3. Springer, 2015.

[27] Demaine, E. D., Hajiaghayi, M., and Klein, P. N. Node-weighted Steiner tree and group Steiner tree in planar graphs. In *Proceedings of the Thirty-sixth International Colloquium on Automata, Languages and Programming: Part I* (2009), Springer, pages 328–340.

[28] Demaine, E. D., Hajiaghayi, M., and Klein, P. N. Node-weighted Steiner tree and group Steiner tree in planar graphs. *ACM Transactions on Algorithms 10*, 3 (2014), 13:1–13:20.

[29] Dhaka Tribune. A road to empires. webpage, 2015. `http://archive.dhakatribune.com/heritage/2014/dec/31/road-empires`. Accessed 15.07.2017.

[30] Diestel, R. *Graph Theory*, fifth edition. Graduate Texts in Mathematics. Springer, 2016.

[31] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 1 (1959), 269–271.

[32] Ding, B., Yu, J. X., Wang, S., Qin, L., Zhang, X., and Lin, X. Finding top-k min-cost connected trees in databases. In *Proceedings of the Twenty-third International Conference on Data Engineering* (2007), IEEE, pages 836–845.

[33] Downey, R. G., and Fellows, M. R. *Parameterized Complexity.* Springer, 2012.

[34] Dreyfus, S. E., and Wagner, R. A. The Steiner problem in graphs. *Networks 1*, 3 (1971), 195–207.

[35] Driscoll, J. R., Gabow, H. N., Shrairman, R., and Tarjan, R. E. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM 31*, 11 (1988), 1343–1354.

[36] Dror, M., Haouari, M., and Chaouachi, J. Generalized spanning trees. *European Journal of Operational Research 120*, 3 (2000), 583–592.

[37] Duin, C., Volgenant, A., and Voss, S. Solving group Steiner problems as Steiner problems. *European Journal of Operational Research 154*, 1 (2004), 323–329.

[38] Erickson, R. E., Monma, C. L., and Veinott Jr, A. F. Send-and-split method for minimum-concave-cost network flows. *Mathematics of Operations Research 12*, 4 (1987), 634–664.

[39] Euler, L. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Petropolitanae 8* (1741), 128–140.

[40] Family search historical records, Various private collections, India. India, hindu pilgrimage records, 1194–2015. webpage, 2016. `https://familysearch.org/wiki/en/India,_Hindu_Pilgrimage_Records_(FamilySearch_Historical_Records)`. Accessed 15.07.2017.

[41] Floyd, R. W. Algorithm 245: Treesort. *Communications of the ACM 7*, 12 (1964), 701.

[42] Flum, J., and Grohe, M. *Parameterized Complexity Theory.* Texts in Theoretical Computer Science. Springer, 2006.

[43] Fomin, F. V., Grandoni, F., and Kratsch, D. Faster Steiner tree computation in polynomial-space. In *Proceedings in the Sixteenth Annual European Symposium on Algorithms* (2008), Springer, pages 430–441.

[44] Fomin, F. V., Grandoni, F., Kratsch, D., Lokshtanov, D., and Saurabh, S. Computing optimal Steiner trees in polynomial space. *Algorithmica 65*, 3 (2013), 584–604.

[45] Fomin, F. V., Kaski, P., Lokshtanov, D., Panolan, F., and Saurabh, S. Parameterized single-exponential time polynomial space algorithm for Steiner tree. In *Proceedings of the Forty-second International Colloquium on Automata, Languages, and Programming* (2015), Springer, pages 494–505.

[46] Ford Jr, L. R. Network flow theory. Technical report, Defense Technical Information Center Document, 1956.

[47] Fredman, M. L., and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM 34*, 3 (1987), 596–615.

[48] Fuchs, B., Kern, W., Mölle, D., Richter, S., Rossmanith, P., and Wang, X. Dynamic programming for minimum Steiner trees. *Theory of Computing Systems 41*, 3 (2007), 493–500.

[49] Fuchs, B., Kern, W., and Wang, X. Speeding up the Dreyfus–Wagner algorithm for minimum Steiner trees. *Mathematical Methods of Operations Research 66*, 1 (2007), 117–125.

[50] Furaih, I. A., and Ranka, S. Memory hierarchy management for iterative graph structures. In *Proceedings of the Twelfth International Parallel Processing Symposium on International Parallel Processing Symposium* (1998), IEEE, pages 298–.

[51] Gabow, H. N. Scaling algorithms for network problems. *Journal of Computer and System Sciences 31*, 2 (1985), 148–168.

[52] Garey, M. R., Graham, R. L., and Johnson, D. S. The complexity of computing Steiner minimal trees. *SIAM Journal on Applied Mathematics 32*, 4 (1977), 835–859.

[53] Garey, M. R., and Johnson, D. S. *Computers and Interactability: A Guide to the Theory of NP-completeness*. W.H. Freeman, 1979.

[54] Garg, N., Konjevod, G., and Ravi, R. A polylogarithmic approximation algorithm for the group Steiner tree problem. In *Proceedings of the Ninth Annual ACM SIAM Symposium on Discrete Algorithms* (1998), SIAM, pages 253–259.

[55] Gentleman, W. M. Row elimination for solving sparse linear systems and least squares problems. In *Proceedings of the Dundee Conference on Numerical Analysis* (1975), Springer, pages 122–133.

[56] Gilbert, E., and Pollak, H. Steiner minimal trees. *SIAM Journal on Applied Mathematics 16*, 1 (1968), 1–29.

[57] Gonnet, G. H. Heaps applied to event driven mechanisms. *Communications of the ACM 19*, 7 (1976), 417–418.

[58] Halperin, E., and Krauthgamer, R. Polylogarithmic inapproximability. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing* (2003), ACM, pages 585–594.

[59] Hauptmann, M., and Karpiński, M. *A Compendium on Steiner Tree Problems*. Institut für Informatik, University of Bonn, 2015. `http://theory.cs.uni-bonn.de/info5/steinerkompendium/netcompendium.pdf`. Accessed 15.07.2017.

[60] He, H., Wang, H., Yang, J., and Yu, P. S. Blinks: Ranked keyword searches on graphs. In *Proceedings of the Thirty-second ACM SIGMOD International Conference on Management of Data* (2007), ACM, pages 305–316.

[61] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, fifth edition. Morgan Kaufmann, 2011.

[62] Hougardy, S., and Prömel, H. J. A 1.598 approximation algorithm for the Steiner problem in graphs. In *Proceedings of the Tenth Annual ACM SIAM Symposium on Discrete Algorithms* (1999), SIAM, pages 448–453.

[63] Hougardy, S., Silvanus, J., and Vygen, J. Dijkstra meets Steiner: a fast exact goal-oriented Steiner tree algorithm. *arXiv preprint arXiv:1406.0492* (2014).

[64] Huddleston, S., and Mehlhorn, K. Robust balancing in B-trees. In *Proceedings of the Fifth GI Conference on Theoretical Computer Science* (1981), Springer, pages 234–244.

[65] Huddleston, S., and Mehlhorn, K. A new data structure for representing sorted lists. *Acta Informatica 17*, 2 (1982), 157–184.

[66] Hwang, F. K., Richards, D. S., and Winter, P. *The Steiner Tree Problem*. Annals of Discrete Mathematics. Elsevier, 1992.

[67] Hwang, F. K., and Weng, J. Hexagonal coordinate systems and Steiner minimal trees. *Discrete Mathematics 62*, 1 (1986), 49–57.

[68] Ihler, E., Reich, G., and Widmayer, P. Class Steiner trees and VLSI-design. *Discrete Applied Mathematics 90*, 1 (1999), 173–194.

[69] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 1: Basic Architecture. White paper, September 2016. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf`. Accessed 15.07.2017.

[70] Intel Corporation. Intel® 64 and IA-32 Architectures Optimisation Reference Manual. White paper, June 2016. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`. Accessed 15.07.2017.

[71] Jarník, V., and Kössler, M. O minimálních grafech, obsahujících n danỳch bod. *Časopis pro pěstování matematiky a fysiky 63*, 8 (1934), 223–235.

[72] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM 24*, 1 (1977), 1–13.

[73] Johnson, E. L. On shortest paths and sorting. In *Proceedings of the ACM Annual Conference - Volume 1* (1972), ACM, pages 510–517.

[74] Jonassen, A., and Dahl, O. Analysis of an algorithm for priority queue administration. *BIT Numerical Mathematics 15*, 4 (1975), 409–422.

[75] Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., and Karambelkar, H. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the Thirty-first International Conference on Very Large Data Bases* (2005), VLDB, pages 505–516.

[76] Karlsson, R. G., and Poblete, P. V. An O(m log log D) algorithm for shortest paths. *Discrete Applied Mathematics 6*, 1 (1983), 91–93.

[77] Karp, R. M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Plenum Press, 1972, pages 85–104.

[78] Karp, R. M. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, 2010, pages 219–241.

[79] Karpinski, M., and Zelikovsky, A. New approximation algorithms for the Steiner tree problems. *Journal of Combinatorial Optimization 1*, 1 (1997), 47–65.

[80] Knuth, D. E. *The Art of Computer Programming: Fundamental Algorithms*, third edition, volume 1. Addison-Wesley, 1997.

[81] Knuth, D. E. *The Art of Computer Programming: Sorting and Searching*, second edition, volume 3. Addison-Wesley, 1998.

[82] Knuth, D. E. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*, volume 4. Addison-Wesley, 2011.

[83] Koch, T., Martin, A. R. D., and Voss, S. Steinlib testdata library. webpage. `http://steinlib.zib.de/steinlib.php`. Accessed 15.07.2017.

[84] Kreher, D. L., and Stinson, D. R. *Combinatorial Algorithms: Generation, Enumeration, and Search*, volume 7. CRC press, 1998.

[85] Kumar, V., and Schwabe, E. J. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing* (1996), IEEE, pages 169–176.

[86] Landahl, H. D. A matrix calculus for neural nets: Ii. *The bulletin of mathematical biophysics 9*, 2 (1947), 99–108.

[87] Landahl, H. D., and Runge, R. Outline of a matrix calculus for neural nets. *The bulletin of mathematical biophysics 8*, 2 (1946), 75–81.

[88] Lappas, T., Liu, K., and Terzi, E. Finding a team of experts in social networks. In *Proceedings of the Fifteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2009), ACM, pages 467–476.

[89] Levinthal, D. Performance Analysis Guide for Intel® Core™ i7 processor and Intel® Xeon™ 5500 processors. Intel Corporation, 2009. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`. Accessed 15.07.2017.

[90] Luce, R. D. Connectivity and generalized cliques in sociometric group structure. *Psychometrika 15*, 2 (1950), 169–190.

[91] Luce, R. D., and Perry, A. D. A method of matrix analysis of group structure. *Psychometrika 14*, 2 (1949), 95–116.

[92] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 01 (2007), 5–20.

[93] Mehta, D. P., and Sahni, S. *Handbook of Data Structures and Applications*. CRC Press, 2004.

[94] Melzak, Z. A. On the problem of Steiner. *Canadian Mathematical Bulletin 4*, 2 (1961), 143–148.

[95] Meyer, U., and Zeh, N. IOefficient undirected shortest paths with unbounded edge lengths. In *Proceedings of the Fourteenth Conference on Annual European Symposium - Volume 14* (2006), Springer, pages 540–551.

[96] Mölle, D., Richter, S., and Rossmanith, P. A faster algorithm for the Steiner tree problem. In *Proceedings of the Twenty-third Annual Symposium on Theoretical Aspects of Computer Science* (2006), Springer, pages 561–570.

[97] Nederlof, J. Fast polynomial-space algorithms using Möbius inversion: Improving on Steiner tree and related problems. In *Proceedings of the Thirty-sixth International Colloquium on Automata, Languages, and Programming: Part I* (2009), Springer, pages 713–725.

[98] NVIDIA Corporation. TESLA K80 GPU accelerator, 2015. `https://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf`.

[99] NVIDIA Corporation. White paper NVIDIA Tesla P100, 2016. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

[100] OpenMP Architecture Review Board. OpenMP Application Programming Interface, Version 4.0. webpage, 2013. `http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`. Accessed 15.07.2017.

[101] Papadimitriou, C. H. *Computational Complexity*. John Wiley, 2003.

[102] Park, J. S., Penner, M., and Prasanna, V. K. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel Distributed Systems 15*, 9 (2004), 769–782.

[103] Prömel, H. J., and Steger, A. *The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity.* Springer, 2002.

[104] Reich, G., and Widmayer, P. Beyond Steiner's problem: A VLSI oriented generalization. In *Proceedings of the Fifteenth International Workshop on Graph-Theoretic Concepts in Computer Science* (1990), Springer, pages 196–210.

[105] Robins, G., and Zelikovsky, A. Tighter bounds for graph Steiner tree approximation. *SIAM Journal on Discrete Mathematics 19*, 1 (2005), 122–134.

[106] Rozenshtein, P., Gionis, A., Prakash, B. A., and Vreeken, J. Reconstructing an epidemic over time. In *Proceedings of the Twenty-second ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016), ACM, pages 1835–1844.

[107] Saavedra, B. R. H. *CPU performance evaluation and execution time prediction using narrow spectrum benchmarking.* PhD thesis, University of California, Berkeley, 1992.

[108] Sarkar, K. M. *The Grand Trunk Road in the Punjab: 1849–1886.* Atlantic Publishers, 1927.

[109] Schrijver, A. On the history of the shortest path problem. *Documenta Mathematica* (2012), 155–167.

[110] Shimbel, A. Structural parameters of communication networks. *The Bulletin of Mathematical Biophysics 15*, 4 (1953), 501–507.

[111] Shoumatoff, A. A reporter at large: The mountain of names. *The New Yorker* (1985), 51.

[112] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 2 (1985), 202–208.

[113] Sozio, M., and Gionis, A. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the Sixteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2010), ACM, pages 939–948.

[114] Tarjan, R. E. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods 6*, 2 (1985), 306–318.

[115] Thejaswi, S. Experimental software implementation of an edge-linear time algorithm for the Steiner problem in graphs, version 1.0, 2017. `https://github.com/suhastheju/steiner-edge-linear`.

[116] Thorup, M. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM 46*, 3 (1999), 362–394.

[117] Thorup, M. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences 69*, 3 (2004), 330–353.

[118] Torricelli, E. *Opera Geometrica*, volume 1/2. Faënza, 1919, pages 90–97.

[119] Torricelli, E. *Opera Geometrica*, volume 3. Faënza, 1919, pages 426–431.

[120] Vazirani, V. V. *Approximation Algorithms*. Springer, 2013.

[121] Vieira, M., and Madeira, H. From performance to dependability benchmarking: A mandatory path. In *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking* (2009), Springer, pages 67–83.

[122] Voss, S. A survey on some generalisations of Steiner's problem. In *Proceedings of the First Balkan Conference on Operational Research* (1990), pages 41–51.

[123] Vuillemin, J. Structures de données. *Notes de cours de l'ecole d'été INRIA* (1975).

[124] Vuillemin, J. A data structure for manipulating priority queues. *Communications of the ACM 21*, 4 (1978), 309–315.

[125] Vygen, J. Faster algorithm for optimum Steiner trees. *Information Processing Letters 111*, 21 (2011), 1075–1079.

[126] Warren, H. S. *Hacker's Delight*, second edition. Addison-Wesley, 2012.

[127] Wei, H., Yu, J. X., Lu, C., and Lin, X. Speedup graph processing by graph ordering. In *Proceedings of the Forty-first ACM SIGMOD International Conference on Management of Data* (2016), ACM, pages 1813–1828.

[128] West, D. B. *Introduction to Graph Theory*, second edition. Prentice Hall, 2001.

[129] Williams, J. W. J. Algorithm 232: Heapsort. *Communications of the ACM 7*, 6 (1964), 347–348.

[130] Zelikovsky, A. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica 9*, 5 (1993), 463–470.